Broadview
www.broadview.com.cn

Apress®

Sencond Edition

The Definitive Guide to SQLite

SQLite 权威指南

第二版

[美] Grant Allen,Mike Owens 著

杨谦 刘义宣 谢志强 译

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

专业人士写给业内人士

SQLite权威指南 第二版

读者朋友们：

企业级计算之外，有一款数据库，它具备大量软硬件需要的关系数据库功能，同时没有传统数据库管理系统的复杂配置、维护，以及高昂的成本。它就是SQLite数据库——一款极其紧凑的可嵌入的数据库，一款能处理巨大数据量的数据库。通过编程和开发，SQLite可以具备大量强大的特性。它也支持C、Java、Perl、PHP、Python、Ruby、Tcl等语言。

《SQLite权威指南（第二版）》包含了SQLite数据库最新版本的全部内容，全面介绍SQLite的功能和API，同时，以SQLite为基础，帮助新人开启数据库开发的第一次旅程。短时间内，您就能开发应用程序，诸如服务器端浏览器插件或者很有流行潜力的iPhone或Android应用！

◎您将会了解到SQLite的C、Java、Perl、PHP、Python、Ruby以及Tcl扩展。

◎您将会掌握SQLite内核。

◎您将会学习使用SQLite开发iOS(iPhone)和Android应用。

全世界范围内，从移动电话和GPS设备到机顶盒与Web浏览器，数以千计的产品选择了SQLite数据库。您可能每天都在不知不觉中使用SQLite。

——Grant Allen和Mike Owens

您将学到以下内容：

★将SQLite部署成iPhone或Android应用的数据库引擎。

★通过Java、C++、PHP以及其他主要语言与SQLite交互。

★充分利用主键约束和外键约束，设计健壮的数据库。

★直接在您的应用程序中嵌入无须维护的数据库。

★通过对SQLite内核的充分理解，改善数据库性能。

★应用数据库特性，例如共享缓存、SQL逻辑测试集、解锁和通知API以及查询计划改善。

本书面向的读者群：

《SQLite权威指南（第二版）》面向需要一个零维护数据库解决方案的开发者，这些开发者将数据库作为一个产品组件嵌入到应用程序中。本书对开发iPhone或Android应用的移动应用开发者也特别有用。对于使用PHP安装自带的数据库的PHP开发者，本书也很有用。

Apress®

策划编辑：张春雨
责任编辑：李利健
封面设计：李　玲

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

上架建议：数据库>移动开发

ISBN 978-7-121-14924-5

9 787121 149245 >

定价：69.00元

Sencond Edition

The Definitive Guide to SQLite

SQLite

权威指南

第二版

[美] Grant Allen,Mike Owens 著

杨谦 刘义宣 谢志强 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

这是一本关于 SQLite 起源、特性、简介、使用、深度解析的书。

全书首先从 SQLite 最初起源、特性、设计理念、实际应用讲解开始，逐步深入、全面地介绍了在各个平台如何使用 SQLite。接着，介绍 SQLite 的一般 SQL 和高级功能的 SQL，采取举例说明，使得本书内容生动有趣。然后，全面介绍了各种语言如何与 SQLite 进行编程交互，重点介绍 SQLite 原生语言 C 语言中的 API，使得使用者可以不管扩展 SQLite 的功能。本书还介绍了目前火热的 IOS 和 Android 开发中如何使用 SQLite，并给出实际例子。最后，介绍了 SQLite 内部架构设计，使得读者可以深入理解 SQLite，高级开发者可以进一步参与 SQLite 开发或者开发自己的 SQLite。

不管您是 SQLite 的初学者，还是 SQLite 资深用户或者是对 SQL 语言和程序设计感兴趣的技术爱好者，都可以从本书汲取营养。

The Definitive Guide to SQLite (2nd Ed.) By Grant Allen, Mike Owens, ISBN: 978-1-4302-3225-4 Original English language edition published by Apress Media. Copyright © 2010 by Apress Media. Simplified Chinese language edition copyright © 2012 by PHEI. All rights reserved.

本书简体中文专有翻译出版权由 Apress Media 授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号　图字：01-2011-5081

图书在版编目（CIP）数据

SQLite 权威指南：第二版 /（美）艾伦（Allen,G.），（美）欧文斯（Owens,M.）著；杨谦，刘义宣，谢志强译. —北京：电子工业出版社，2012.1

书名原文：The Definitive Guide to SQLite (2nd Ed.)

ISBN 978-7-121-14924-5

Ⅰ. ①S… Ⅱ. ①艾… ②欧… ③杨… ④刘… ⑤谢… Ⅲ. ①SQL 语言－程序设计 Ⅳ. ①TP311.132

中国版本图书馆 CIP 数据核字(2011)第 221911 号

策划编辑：张春雨

责任编辑：李利健

印　　刷：北京丰源印刷厂

装　　订：三河市鹏成印业有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱　邮编：100036

开　　本：787×980　1/16　印张：25.25　字数：485 千字

印　　次：2012 年 1 月第 1 次印刷

定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

谨以此书献给我的母亲，

感谢她对我无尽的支持！

译者序

您用过 FireFox（火狐)浏览器吗？您用过金山词霸、iPhone、iPad 或 Android 系统吗？如果答案是 Yes，说明您已经是 SQLite 用户或者从 SQLite 中受益了。SQLite 作为一款嵌入式关系数据库，应用十分广泛，已经被大量应用在许多基础的软件中，并成为这些基础软件的组成部分。例如，您手中的 iPhone 或 Android 手机，您电脑上的浏览器或者其他软件，您访问的用 PHP 编写的网站等。

虽然 SQLite 被越来越广泛地应用，但是一直缺乏一本全面、深入浅出地介绍 SQLite 的书，本书就是这样一本从零开始不断深入探究 SQLite 的书。

本书特色鲜明，它不仅为开发人员提供了全新的数据库选择，并且全面介绍了如何应用这种选择。全书第 1、2 章全面细致且重点突出地介绍了 SQLite 的起源和特点，并与主流大型关系数据库进行比较，这可作为数据库选型的重要参考。本书也是嵌入式关系数据库研究人员的知音，因为它介绍了嵌入式数据库中最典型的代表 SQLite，并对 SQLite 内核做了简要介绍。最重要的是，本书是 SQLite 相关开发者的重要帮手。从第 3 章开始，全面介绍 SQLite 中的 SQL、高级 SQL、核心 API 和扩展 API 以及各种语言扩展、iPhone 和 Android 开发中如何应用 SQLite 等。内容翔实，实例丰富，理论与实践并举。

译者在翻译本书[①]时，一边是感到万分的荣幸，一边也是战战兢兢。吉尔伯特·海特（美国教育家）曾经说过：写了一本很糟糕的书只是犯错而已，而把一本好书翻译得很糟糕则是犯罪。但是对于本书这样最佳的 SQLite 理论与实践之作，即便是冒着犯罪的风险，也值得翻译出来并推荐给大家。

① 为了便于读者对照原书阅读，本书正文页边距处用“⇨”标出了原书页码。本书的索引所列页码为原英文版页码。

本书承载的内容丰富而深入。在翻译的过程中，译者也常感汗颜，原来自诩对 SQLite 颇为了解的我深刻感受到自己知识面的浅薄和不完整，于是翻译过程也成了自己检讨和学习的过程，收获颇丰。本书的作者是 SQLite 方面的知名研究者与实践者，其中一位是 PySQLite 的创建者。他们有着丰富的 IT 从业经验，书中的很多内容都是他们长年实践的经验和心得。因此，希望大家反复阅读，必定受益无穷。

由于本书作者的知识广博，经验丰富，而译者本人知识有限、时间仓促，难免对作品的个别之处无法原汁原味地呈现给读者，在此敬请各位读者多多包涵，不吝赐教，指出纰漏。我也希望能就本书的内容与读者有更多的交流，我的邮箱是 yqbjtu@live.com。

最后，我要感谢电子工业出版社博文视点公司的张春雨编辑在译书的过程中给予我们的莫大帮助。我还要感谢我的朋友雷昕、李晓亮、杨艳、刘争涛、吴林、杨建轩、尹相乐、季旻、房华、张晓光和东莞经济贸易学校的谢志强老师等。特别感谢北京恩墨科技有限公司技术总监、Itpub Oracle 数据库管理版与 Oracle 高可用版版主张乐奕（网名 kamus），SQLite 中文社区（http://www.sqlite.com.cn/）站长林轩，原 Oracle 亚洲研发中心高级 DBA，玛氏信息技术服务有限公司亚太区数据库运营经理叶熙昌，他们的参与使得本书的质量得到了进一步的保证。

杨　谦

2011 年 8 月于北京志新村

推荐序 1

SQLite——一个被广泛使用的嵌入式关系数据库，在短短的几年时间里飞速发展。大家对 SQLite 已经不再陌生，越来越多的人开始学习它、使用它，并且爱上它。记得 2005 年第一次在项目中使用 SQLite 时，它还仅仅是一个新兴的开源项目，那一年它获得了 Google 和 O'REILLY 的年度开源大奖。虽然那时的 SQLite 功能还不是很完备，稳定性上也还有些许瑕疵，但是它小巧的身姿、极高的运行效率以及广泛的平台支持，还是深深地吸引了我。之后我便一发不可收拾，在许多开发的软件项目中都自然而然地用到了它。当然，事实证明我的选择是正确的，如今无数的软件巨头如 Google、Apple、Adobe、Mozilla 等也都在自己的产品中开始使用 SQLite，并且积极地为 SQLite 开发社区提供技术和资金等方面的支持，从而确保了社区能够拥有持续的活力。我相信，随着以 SQLite 作为底层存储引擎的 Android 操作系统的日益流行和其他移动领域开发的增多，今后会有更多的开发人员投入到 SQLite 的相关开发中来。

然而，一直以来困扰所有开发者的是，缺乏一本深入讲解 SQLite 的书籍，无论是入门者还是希望更深入了解的人，只能寄托于在网上搜索相关资料，这样无疑会增加学习的成本。为此，我建立了 SQLite 中文社区，希望能让所有的开发人员更方便地找到所需资料。本书作为 SQLite 经典著作的第二版，其第一版发表于 2006 年，书中详细而精辟地讲述了有关 SQLite 的开发技术，内容可谓深入到各个方面。在新版中，作者特意加入了 SQLite 在 iOS 和 Android 系统中的开发内容，从而为 Mobile 领域的开发人员学习 SQLite 提供了帮助。今天，在很多人的辛勤付出之下，本书的中文版终于能够与读者见面，这对于国内的 SQLite 开发者无疑是一件非常值得庆祝的事。

作为一名普通的 SQLite 爱好者，能够参加中文版的审阅工作，使我有幸在第一时间看到了本书的中文版。在此，我对译者的翻译功底深表钦佩，其深厚的技术功底跨越

了语言障碍，完整而准确地将原书的内容表达无遗。此外，中文版的排版行文风格也与原书做到了高度的统一。因此，在阅读之后，我非常高兴地向大家推荐这本书，希望它能够帮助大家跨入 SQLite 的世界，领略它轻巧而灵动的美。

SQLite 中文社区站长　林轩

推荐序 2

“您可能是一名资深的 SQLite 用户或者开发者，也可能阅读这本书是您第一次走进 SQLite 世界。不论您具有何种背景，每天都在不知不觉中使用着 SQLite。因为 SQLite 是一款被广泛应用于移动设备的数据库，从 iPhone、iPod 到最新的平板电脑、iPad、媒体播放器等。”

这是本书第 9 章的开篇语，正如您看到的，在智能移动设备被炒得火热并且确实在领导今后科技发展潮流的今天，哪怕本书仅仅只有第 9、10 章，对于立志于在移动开发领域有所涉猎的您来说，这本书都是值得购买的。

Itpub Oracle 数据库管理版与 Oracle 高可用版版主

北京恩墨科技有限公司技术总监　**张乐奕**

推荐序 3

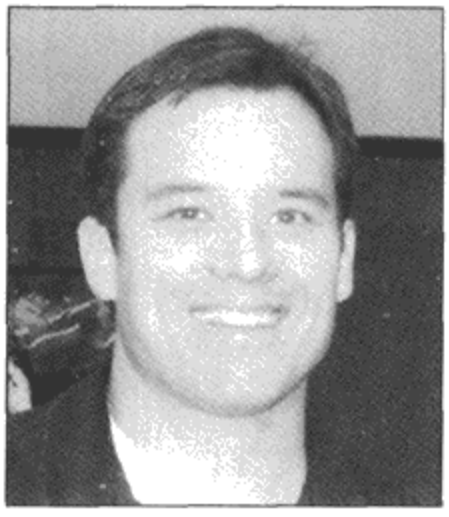
其实不用我多说，大家也知道现在移动开发有多火。君不见大街小巷，男男女女，老老少少手上都有一部 iPhone、iPad 或 Android 手机，这些移动设备已经如潮水般地涌入我们的日常生活。iOS 和 Android 作为一种移动设备操作系统，自然也有对应的数据库管理数据，而这个数据库就是本书介绍的 SQLite。SQLite 作为一种便携式轻量级数据库，非常适合移动开发，有志于从事这个行业的人都应该了解如何使用 SQLite 开发出健壮的移动应用程序。另外，SQLite 是完全开源的数据库，那些对数据库底层引擎有兴趣的朋友们，相信读完此书会有获益匪浅的感受。

Oracle 10g OCM，玛氏信息技术服务有限公司亚太区数据库运营经理　**叶熙昌**

关于作者



Grant Allen 具备 20 多年的 IT 工作经验，做过软件开发公司的首席技术官（CTO）和 Google 的数据架构师。他的工作横跨多个行业，有政府部门和学术部门，大型系统设计、开发、性能、创新以及破坏性改变的咨询。Grant 也是学术会议和行业会议的常客，他经常做一些关于数据挖掘、协作技术、关系数据库以及技术业务的报告。Grant 在高新技术公司领导破坏性创新理论项目的业余时间取得了他的博士学位。



Mike Owens 是得克萨斯州沃斯堡一家大型房地产公司的 IT 主管，他负责开发和管理公司的核心系统。之前他在橡树岭国家实验室（该实验室是美国能源部所属的一个大型国家实验室）担任流程设计工程师，在 Nova 信息系统公司当过 C++程序员。他是 PySQLite 的最初创建者，PySQLite 是 SQLite 的 Python 扩展。Mike 毕业于田纳西大学诺克斯维尔分校，获得化学工程学士学位。

Mike 喜欢慢跑、弹吉他、滑雪，以及与同伴们一起在得克萨斯州狭长的地带上打猎。他与妻子、两个女儿以及两只小猎狗在得克萨斯州沃斯堡生活。

关于技术评审



D. Richard Hipp 是 SQLite 和 Fossil DVCS 项目的创建者和项目组长。Richard 和他选中的一小部分人员全职为一个国际客户维护这些产品并增强其功能。

Richard 出生于北卡罗来纳州夏洛特，目前他与妻子 Ginger 在那里生活。Richard 于 1984 年获得乔治亚理工大学的电机工程硕士学位，1992 年获得杜克大学博士学位。

致　谢

向 Apress 出版社的团队表示我的谢意，特别是我的编辑 Jonathan Gennick 和项目经理 Jennifer Blackwell，有了他们“The Definitive Guide to SQLite”这本书就像是自动写完的！好吧，不开玩笑了，但毫无疑问的是——他们使得本书的写作更加愉快而有意义。

特别感谢 D.Richard Hipp，不仅感谢他创建了 SQLite，也要感谢他对手稿的无数次审阅与修订。他告诉我很多 SQLite 最新和最重要的特性，希望我能正确地在本书体现这些内容。

最后，感谢我的朋友和家人，他们容忍我再写一本书，以及写作时的疯狂、滑稽。

简 介

《SQLite 权威指南》第二版全面介绍了 SQLite 和 SQLite 相关的知识，以及多种情况下使用 SQLite 的经验。不管您是坚定的 C 开发者、移动设备爱好者，还是想知道最好的嵌入式小型数据库引擎是如何开发的，本书都适合您。

先决条件

本书假设您之前没有 SQLite 知识，但是不管哪种水平的人都能从中受益。SQLite 是用 C 语言编写的，带有扩展的 C API，也支持很多其他语言，例如 Python、Tcl、Ruby 和 Java。作为数据库引擎，它也使用 SQL。虽然本书的例子可以使具有各种水平的读者受益，但是限于篇幅，不在这里讲授这些语言。

本书组织结构

本书由 11 章组成，主要包含以下内容：

- SQLite 介绍、获取和安装。
- 使用 SQLite 中的 SQL。
- SQLite 中的 C API。
- 其他语言例如 Python、Tcl、Ruby 和 Java 中如何使用 SQLite。
- 移动设备开发中如何使用 SQLite。
- SQLite 内核与新特性。

您可以随便跳到自己喜欢的章节，但是第 5～7 章都是与 C API 相关的，最好按顺

序阅读。

如何获取示例的源代码

本书示例中的所有源代码都可从 Apress 网站（http://apress.com/book/view/1430232250）中下载。在“Book Resources”框中单击“Source Code”链接即可。

目 录

译者序 IV
推荐序 1 VI
推荐序 2 VIII
推荐序 3 IX
关于作者 XX
关于技术评审 XXI
致谢 XXII
简介 XXIII

第 1 章 SQLite 介绍 1

嵌入式数据库 1
开发者的数据库 2
管理员的数据库 4
SQLite 历史 4
谁在使用 SQLite 5
体系结构 6

接口 6
编译器 7
虚拟机 7
后端 9
工具和测试代码 9

SQLite 的特性和设计理念……10
零配置……10
移植性……10
紧凑性……11
简单性……11
灵活性……12
自由授权……12
可靠性……12
易用性……12
性能和限制……13
本书面向的读者……16
本书的组织结构……17
附加说明……18
总结……19
第 2 章　入门……21
何处获取 SQLite……21
Windows 上的 SQLite……22
获得命令行程序……22
获取 SQLite 动态链接库（DLL)……25
在 Windows 上编译 SQLite 源代码……26
使用微软 Visual C++构建 SQLite DLL……29
用 Visual C++构建动态链接 SQLite 的客户端……31
用 MinGW 构建 SQLite……32
Linux、苹果 Mac OS X 以及其他 POSIX 系统上的 SQLite……34
二进制和包……34
从源代码编译……35
命令行程序……37
Shell 模式下的 CLP……37
命令行模式的 CLP……39

数据库管理 40
创建数据库 40
获得数据库的 Schema 信息 42
导出数据 43
导入数据 44
格式化 45
导出带分隔符的数据 46
执行无人值守维护 46
备份数据库 47
获得数据库文件的信息 48
其他 SQLite 工具 50
总结 51

第 3 章　SQLite 中的 SQL 53

数据库示例 53
安装 55
运行示例 55
语法 56
命令 58
常量 58
关键字和标识符 59
注释 59
创建数据库 59
创建表 60
修改表 61
数据库查询 62
关系操作 62
select 命令与操作管道 63
过滤 66
限定和排序 71

函数（Function)和聚合（Aggregate)……73
分组（Grouping)……74
去掉重复……79
多表连接……79
名称和别名……85
子查询……87
复合查询……89
条件结果……91
处理 SQLite 中的 Null……93
总结……95
第 4 章　SQLite 中的高级 SQL……97
修改数据……97
插入记录……97
更新记录……101
删除记录……102
数据完整性……102
实体完整性……103
域完整性……108
存储类……113
视图……116
索引……118
触发器……120
事务……124
事务的范围……124
冲突解决……125
数据库锁……128
死锁……129
事务的类型……130
数据库管理……131

附加数据库 131
数据库清理 133
数据库配置 133
系统目录 137
查看查询计划 137
总结 138
第 5 章 SQLite 设计与概念 139
API 140
主要数据结构 140
连接和语句 141
核心 API 142
可操作的控制 150
使用线程 151
扩展 API 151
创建用户自定义函数 151
创建用户自定义聚合 152
创建用户自定义排序 153
事务 153
事务生命周期 154
锁状态 154
读事务 156
写事务 157
调整页面缓存 160
过渡到独占状态 161
调整页面缓存 161
等待锁 162
繁忙处理 162
使用恰当的事务 164
代码 165

使用多个连接 ····· 165
finalize()函数的重要性 ····· 167
共享缓存模式 ····· 168
总结 ····· 168

第 6 章　核心 C API ····· 171

查询封装 ····· 171
连接与断开连接 ····· 172
执行查询 ····· 174
获取表查询 ····· 178
查询准备 ····· 179
编译 ····· 180
执行 ····· 181
完成与重置 ····· 182
获取记录 ····· 184
获取字段信息 ····· 184
获取字段值 ····· 186
一个实例 ····· 187
查询参数化 ····· 189
参数编号 ····· 192
参数命名 ····· 192
Tcl 参数 ····· 193
错误与异常 ····· 194
错误处理 ····· 194
繁忙情况处理 ····· 196
模式改变处理 ····· 197
操作控制 ····· 198
提交钩子 ····· 198
回滚钩子 ····· 199
更新钩子 ····· 199

授权函数 200
线程 210
共享缓存模型 210
线程与内存管理 213
总结 214
第 7 章　扩展 C API 215
API 216
注册函数 217
步骤函数 218
返回值 219
函数 220
返回值 222
数组与内存清理器 223
错误处理 224
返回输入值 224
聚合 225
注册函数 226
实例 226
排序规则 230
排序法定义 231
简单例子 234
按需排序 237
总结 238
第 8 章　语言扩展 239
选择一种扩展语言 240
Perl 242
安装 242
连接 243
查询处理 243

参数绑定 ····· 245
用户自定义函数 ····· 246
聚合 ····· 247
Python ····· 248
安装 ····· 248
连接 ····· 249
查询处理 ····· 249
参数绑定 ····· 251
用户自定义函数 ····· 253
聚合 ····· 253
APSW（另一种 Python 接口） ····· 254
Ruby ····· 255
安装 ····· 255
连接 ····· 256
查询处理 ····· 256
参数绑定 ····· 257
用户自定义函数 ····· 259
Java ····· 260
安装 ····· 260
连接 ····· 261
查询处理 ····· 262
自定义函数和聚合 ····· 264
JDBC ····· 266
Tcl ····· 268
安装 ····· 268
连接 ····· 268
查询处理 ····· 269
用户自定义函数 ····· 272
PHP ····· 272
安装 ····· 273

连接 273
查询 274
用户自定义函数和聚合 276
总结 278

第 9 章 iOS 开发中的 SQLite 279
SQLite iOS 开发的先决条件 279
注册成为 Apple 软件开发者 280
下载并安装 Xcode 和 iOS SDK 280
其他开发环境 283
建立 iSeinfeld iOS SQLite 应用 284
第一步：创建一个新的 Xcode 工程 285
第二步：将 SQLite 框架添加到工程 285
第三步：准备 Foods 数据库 287
第四步：为食品数据创建类 289
第五步：访问和查询 SQLite 数据库 293
第六步：最后包装和配置 iSeinfeld 应用程序 297
运行 iSeinfeld 297
iOS 中处理大型 SQLite 数据库 299
总结 300

第 10 章 Android 开发中的 SQLite 301
SQLite Android 开发的先决条件 301
检查先决条件和 JDK 302
下载和安装 Android SDK Starter 包 302
下载和安装 Android 开发工具 303
添加 Android 平台和组件 304
Android SQLite 类和接口 306
使用基础帮助类：SQLiteOpenHelper 307
SQLiteDatabase 类 308
在实际中应用 SQLiteOpenHelper 和 SQLiteDatabase 312

使用 SQLiteQueryBuilder 类查询 SQLite 数据库……315
搭建 Seinfeld Android SQLite 应用程序……317
创建新的 Android 工程……318
将 Seinfeld SQLite 数据库添加到工程……319
查询 Food 数据库表……319
定义用户接口……320
连接数据和用户接口……321
查看完成的 Seinfeld 应用程序……322
SQLite Android 应用程序的注意事项……322
Android 数据库的备份……323
Android 系统处理大型 SQLite 数据库……324
总结……324

第 11 章　SQLite 内部机制及新特性……325

B-tree 和 Pager 模块……325
数据库文件格式……325
B-tree API……330
显示类型、存储类以及亲缘性介绍……332
显示类型……333
类型亲缘性……335
亲缘性和存储……336
执行中的亲缘性……336
预写日志……340
WAL 工作原理……341
激活和配置 WAL……342
WAL 的优缺点……343
启用 WAL 时 SQLite 数据库的操作问题……343
总结……345

索引……347

第 1 章

SQLite 介绍

SQLite 是一款开源的、嵌入式关系型数据库。它最初发布于 2000 年，用来省去专用于关系型数据库管理系统日常费用，在此前提一下同样可以为应用软件提供一种方便的数据管理方式。SQLite 在便携性、易用性、紧凑性、高效性和可靠性方面有突出的表现。

嵌入式数据库

SQLite 是一款嵌入式数据库，它没有独立运行的进程，它与所服务的应用程序在应用程序进程空间内共生共存。它的代码与应用程序代码也是在一起的，或者说嵌入其中，作为托管它的程序的一部分。对外部观察者而言，无法明确看到这样的程序有一个关系型数据库管系统（RDBMS）在运行。程序只需要做自己的事，管理自己的数据，不需要详细了解 SQLite 是如何工作的。但是在其内部有一个完整的、自我包含的数据库引擎在工作。

数据库服务器在程序中的好处是不需要网络配置或管理。数据库服务器和客户端运行在同一个进程中。可以想象一下这样做解放了哪些工作：不用担心防火墙或者地址解析，不用浪费时间管理复杂的授权和权限。客户端和服务器端运行在同一个进程中，可以减少网络调用相关的消耗，简化数据库管理，使程序更容易部署。所需要做的只是将其正确地编译到程序中。

从图 1-1 中可知，一个 Perl 脚本、一个标准 C/C++程序和一个 Apache 托管的 PHP 脚本都使用 SQLite。Perl 脚本导入 DBI::SQLite 模块，并通过它链接到 SQLite C API，进入 SQLite 运行库。PHP 运行库与 C++程序工作相似。总之，这三种处理方式都要与 C API 交互。因此，这三种都有嵌入在它们进程空间的 SQLite，这样做不仅可以使这些

进程运行各自的代码，也可以成为独立的数据库服务器。此外，尽管每个进程中都有独立的数据库服务器，但它们依然可以操作同一个数据库文件，因为 SQLite 利用操作系统功能来完成数据的同步和加锁。

目前市场上有多种为嵌入式应用设计的关系型数据库产品，例如 Sybase SQL Anywhere、Oracle TimesTen 和 BerkleyDB、Pervasive PSQL，以及微软的 Jet Engine。有些厂家将他们的大型数据库产品翻新，开发出嵌入式的变种，例如，IBM 的 DB2 Everyplace、Oracle 的 Database Lite 和微软的 SQL Server Express。开源数据库 MySQL 和 Firebird 也提供了嵌入式版本。所有这些产品中，只有一款是开源的，不受许可证费用约束，而且是专门为嵌入式设计的——这就是 SQLite。

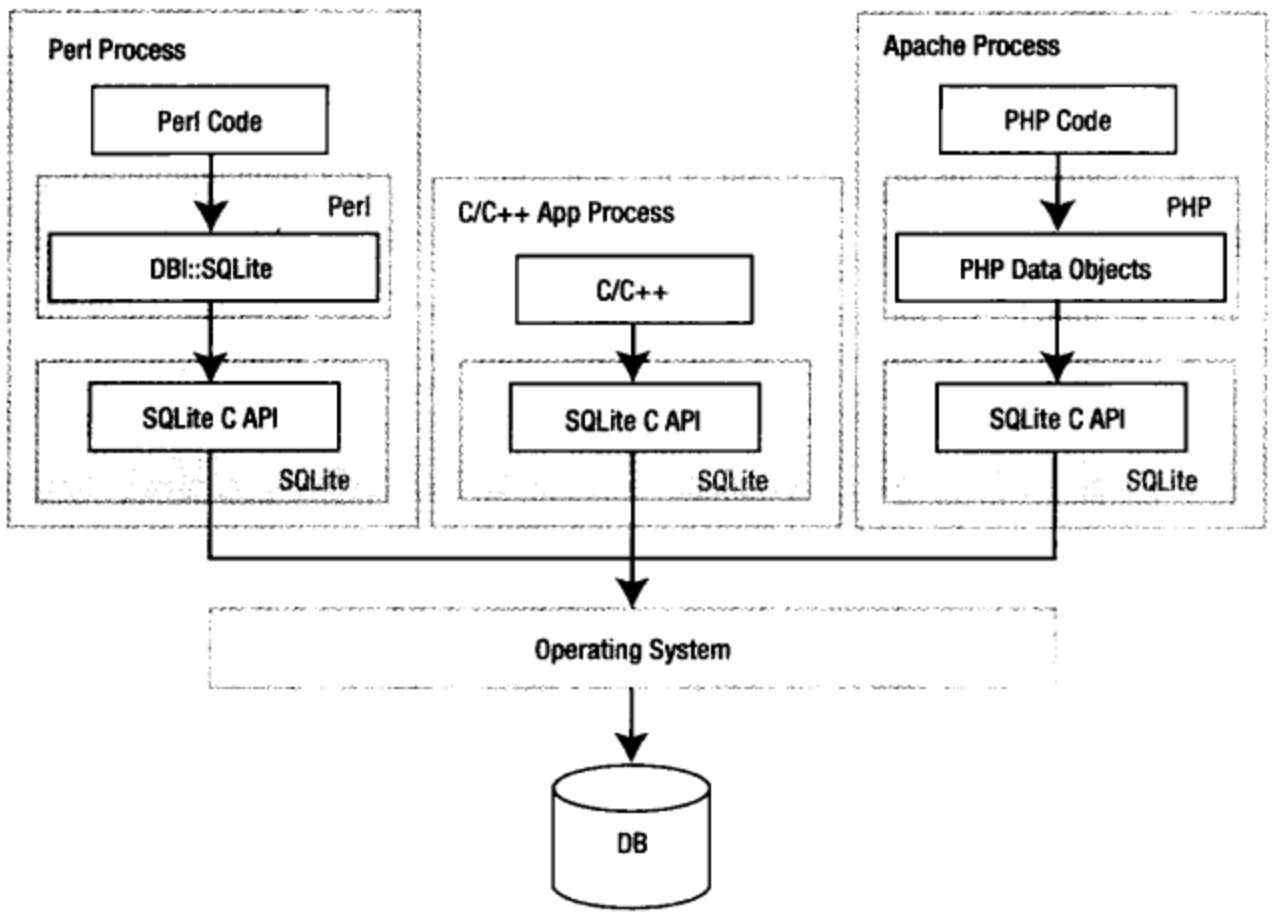


图 1-1　内嵌在主进程中的 SQLite

开发者的数据库

SQLite 用途十分广泛。它是数据库、编程库、命令行工具，也是学习关系型数据库的优秀工具。它的使用方式多种多样——嵌入式环境、网站、操作系统服务、脚本和应

用程序。对程序员来说，SQLite 就像一个数据传送带，它提供了一种方便的绑定应用程序与其数据的方法。可以说，SQLite 的前途不可限量。在 Web 环境中，SQLite 可以帮助管理复杂的会话信息，而不用将这些会话信息序列化成数据存储在 blob 中，可以从单独的会话数据库中读取和写入单个会话信息。SQLite 也可以用来替代开发测试中的关系数据库——不再需要外部的关系型数据库管理系统或者网络配置，也不再需要程序员操心用户名和密码。SQLite 也可以作为一个缓存，用于存储配置数据，或者可以作为应用软件的文件格式，因为它采用跨平台兼容的二进制。

除了作为存储容器之外，SQLite 还可以作为一个通用的数据处理工具。由于 SQLite 本身较小，并且复杂性较低，因此，使用 SQLite 可以很容易地将一些应用程序数据结构转化为表，并保存在内存数据库中。对于很多开发者、分析师以及其他熟悉关系数据库和 SQL 的人而言，不必自己写算法来对数据结构进行操作和排序，使用这种方法就可以操作互相关联的数据，可以完成很繁重的任务。如果您是程序员，想象一下在程序中实现如下 SQL 语句功能需要多少代码：

```
SELECT x, STDDEV(w)
FROM table
GROUP BY x
HAVING x > MIN(z) OR x < MAX(y)
ORDER BY y DESC
LIMIT 10 OFFSET 3;
```


如果您已经熟悉 SQL，想象一下，以熟悉的编程语言编码实现同等的子查询、组合查询、排序子句或者多个连接，需要多少工作。SQLite 以最小的代价将这些功能嵌入在应用程序中。数据库引擎直接集成到代码中，您可以将 SQL 看成是离线引擎，通过它可以在程序中实现复杂的排序算法。随着数据集的不断增大，这种方式的优势越来越明显，因为如果自己写代码，算法会越来越复杂。更好的是，SQLite 可以配置使用固定大小的内存，如果数据大小超过了指定限制，可以将数据存放到磁盘。如果自己写算法，实现这一功能会更困难。使用 SQLite，通过一个 SQL 命令就可以实现这一功能。

对程序员而言，SQLite 还是一个很好的学习程序设计的工具——研究计算机科学主题的聚宝盆。词法分析器、语法分析器、虚拟机、B-tree 算法、缓存、程序架构等，这是探究很多计算机科学经典概念非常好的方式。SQLite 的模块化、小型化和简易性使得它很容易将一个个主题作为单独学习的例子，任何人都可以较快地理解。

管理员的数据库

SQLite 不仅是程序员的数据库，对系统管理员而言，它也很有用。它短小、紧凑而精致，就像一些常用工具，如 find、rsync 和 grep。SQLite 提供了可在 Shell 脚本内使用的命令行工具。而且，它能与大量的脚本语言非常协调地工作，例如 Perl、Python、TCL 以及 Ruby。综合这两者有助于完成各种任务，例如日志文件数据的聚合、监视磁盘配额，或者在共享网络中执行带宽计费。此外，因为 SQLite 数据库是普通的磁盘文件，它们很容易操作、传输和备份。

对管理员而言，SQLite 是学习关系型数据库的便捷工具。它是开始学习关系数据库概念和实际实现的理想选择。它能很快且很容易地安装在常见的操作系统中，它的数据库文件可以自由共享且无须任何转换。它具有关系型数据库的各种功能而又不令人生畏。它的程序和数据库文件仅用存储芯片或者 U 盘就可携带。

SQLite 历史

从某种程度上说，SQLite 最初的构思是在一条军舰上进行的。当时在通用动力工作的 SQLite 的作者 D. Richard Hipp 正在为美国海军编制一种在导弹驱逐舰上使用的程序。那个程序最初运行在 Hewlett-Packard UNIX（HPUX）上，后台使用 Informix 数据库。对那个具体应用而言，Informix 有点儿太强大了。一个有经验的数据库管理员（DBA）安装或升级 Informix 可能需要一整天，如果是没经验的程序员，这个工作可能永远也做不完。那个程序真正需要的只是一个自我包含的数据库，它易使用并能由程序控制传导，另外，不管其他软件是否安装，它都可以运行。

2000 年 1 月，Hipp 开始和一个同事讨论关于创建一个简单的嵌入式 SQL 数据库的想法，这个数据库将使用 GNU DBM 哈希库（gdbm）做后台，同时这个数据库将不需要安装和管理支持。后来，一有空闲时间，Hipp 就开始实施这项工作，2000 年 8 月，SQLite 1.0 版发布了。

按照原定计划，SQLite 1.0 用 gdbm 作为存储管理器。然而，Hipp 不久就用自己实现的能支持事务和记录按主键存储的 B-tree 替换了 gdbm。随着第一次重要升级的进行，SQLite 有了稳定的发展，功能和用户也在增长。2001 年中期，很多项目——开源的或商

业的——都开始使用 SQLite。在随后的几年中，开源社区的其他成员开始为他们喜欢的脚本语言和程序库编写 SQLite 扩展。一个接着一个，既 Perl、Python、Ruby、Java 和其他主流的程序设计语言的扩展之后，新的扩展如 SQLite 的 ODBC 接口出现并证明了 SQLite 的广泛应用和实用功能。

2004 年，SQLite 从版本 2 升级到版本 3，这是一次重大升级。这次升级的主要目标是增强国际化，支持 UTF-8、UTF-16 及用户定义字符集。虽然 3.0 版最初计划在 2005 年夏季发布，但美国在线提供了必要的支持，希望其务必在 2004 年 7 月发布。除国际化功能外，版本 3 也带来其他很多新特性，例如修订过的 C API、更紧凑的数据库文件格式（比原来节省 25%的空间）、弱类型、二进制大对象（BLOB）的支持、64-bit 的 ROWID、自动清理未使用空间和改进了的并发控制等。除了这些新特性，总的程序库依然小于 240KB，而此时大部分家用 PC 的内存都开始以吉字节（GB）计算了。版本 3 的另一个改善是代码清理——重新审视并重写了代码，丢弃了 2.x 系列中堆积的无关元素。

SQLite 持续增加新特性并依然坚守其最初的设计目标：简单、灵活、紧凑、快速和整体上的易用。本书出版时，SQLite 考虑增加递归触发器、帮助优化器产生更快查询的分布式直方图，操作系统能支持的异步 I/O，再下面是什么了？视情况而定。也许您或者您的公司将赞助下一个大的特性，使得这个小数据库更好、更强大。

谁在使用 SQLite

如今，SQLite 已经被多种软件和产品使用。它被用在苹果的 Mac OS X 操作系统、Safari 的 Web 浏览器、Mail.app 的电子邮件程序、RSS 的管理器，以及苹果的 Aperture 照片软件中。也许，苹果对 SQLite 的最大应用在 iPhone 时代。您可以在 iPhone 中发现很多应用程序，例如联系人数据库、Notes 等，它们都依赖 SQLite 数据库。这种依赖也扩展到 iPad。第 9 章我们会回到这一话题，详细讨论苹果移动平台上的 SQLite。

也可以在 Sun 公司的 Solaris 操作系统中找到 SQLite，特别是在 Solaris 10 中首次亮相的服务管理工具，一种预先自我修复技术的核心组件，它是由数据库支撑的。Mozilla 项目 mozStorage C++/JavaScript API 层也在使用 SQLite，该层是火狐浏览器、Thunderbird 以及 Sunbird 存储个人信息的主要方式。SQLite 也被添加成为 PHP 5 标准库的一部分。也随同诺基亚公司跨平台的 Qt C++应用程序框架一起发布，Qt C++应用程序框架是流行的 KDE 窗口管理器和许多其他应用软件的基础。SQLite 在嵌入式平台特别流行。

Richard Hipp 的 SQLite 相关业务中很多都是将 SQLite 移植到各种专有的嵌入式平台上。Symbian 使用 SQLite 在原生 Symbian OS 平台上提供 SQL 支持。谷歌也在 Android 移动手机操作系统和用户空间应用程序中广泛使用 SQLite。SQLite 在 Android 平台上的应用如此普遍，第 10 章会专门讨论如何在 Android 设备上使用 SQLite。SQLite 也包含在一些商业开发的手机应用产品中。

5 尽管 SQLite 很少做广告，但它还是用在了多种消费类产品中，因为一些懂技术的消费者还是在底层发现了它。例如，D-Link 的 Media Lounge、Slim 设备 Squeezebox 音乐播放器以及飞利浦 GoGear 个人音乐播放器，甚至一些聪明的消费者在 Complete New Yorker DVD 集中发现了嵌入其中的 SQLite 数据库，Complete New Yorker DVD 是《New Yorker》杂志的所有发行版的数字化库——很明显，随着搜索软件而被使用的。

可以在很多开源项目中发现 SQLite 的身影，例如 Yum（Fedora Core 的包管理器）、Movable Type、DSPAM、Edgewall 软件非常优秀的 Trac SCM 和项目管理系统的后端存储工具；可能最知名的是，SQLite 被用做 Mozilla 火狐浏览器的内嵌数据库，甚至可以在其他开源项目中找到 SQLite 的核心工具。例如 Lemon 解析产生器，lighttpd 网络服务器项目用它来产生读取配置文件的解析代码。实际上，看起来正是各种这样的应用使得谷歌注意到并授予 Richard Hipp 2005 年 O'REILLY 开源会议“最佳集成奖”，这也是谷歌以 Android 进入移动领域前好几年的事，读者可进入 www.sqlite.org/famous.html 网站查看一些 SQLite 重要用户的其他想法。

体系结构

SQLite 拥有一个简洁的、模块化的体系结构，并引进了一些独特的方法进行关系型数据库管理。它由可以划分为 3 个子系统的 8 个独立模块组成，如图 1-2 所示。这些模块将查询过程划分为几个独立的任务，就像在流水线上工作一样。在体系结构栈的顶部编译查询语句，在中部执行，在底部处理存储并与操作系统交互。

6 接口

接口处于栈的顶端，由 SQLite C API 组成。程序、脚本语言还有与 SQLite 交互的库文件最终都是通过它与 SQLite 交互的。从字面上理解，这里是开发者、管理员、学

生或者痴迷的科学家与 SQLite 通信的地方。

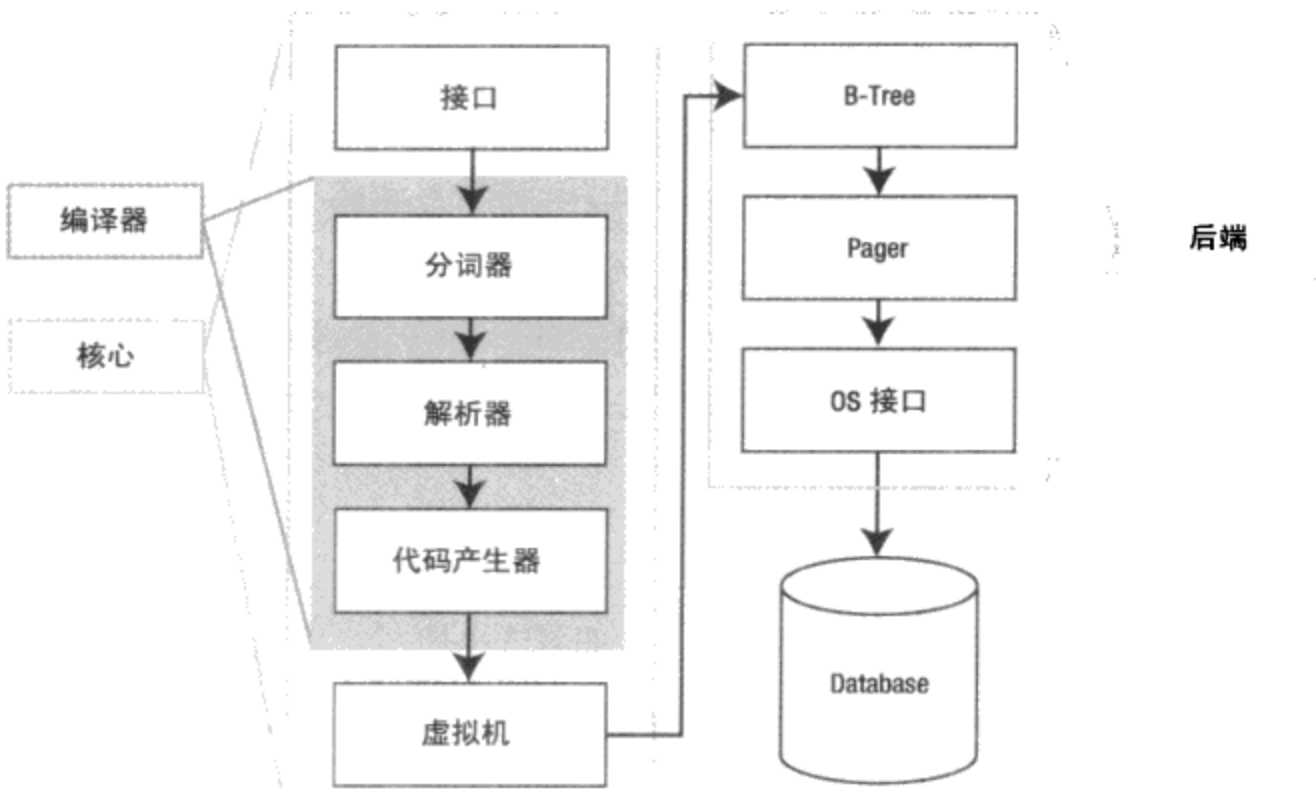


图 1-2　SQLite 体系结构

编译器

编译过程从词法分析器（Tokenizer）和语法分析器（Parser）开始。它们协同处理文本形式的结构化查询语句（Structured Query Language，SQL），分析其语法有效性，然后转化为底层能更方便地处理的层次化数据结构。SQLite 词法分析器是手动编码实现的，它的语法分析器是由 SQLite 特定的语法分析生成器 Lemon 产生的。Lemon 语法分析生成器采取了一些特殊的预防措施防止内存泄漏，并且性能较高。SQL 语句先被分解为一个个词法记号，经过评估后以语法树形式重组，然后语法分析器将该树下传给代码生成器。

代码生成器将语法树翻译成一种 SQLite 专用的汇编代码，这些汇编语言由一些最终由虚拟机执行的指令组成。代码生成器的唯一工作是将语法树转换为完全由这种汇编语言编写的微程序并交给虚拟机处理。

虚拟机

架构栈的中心部分是虚拟机，也叫做虚拟数据库引擎（Virtual DataBase Engine，

VDBE）。VDBE 是基于寄存器的虚拟机，在字节码上工作，这使得它可以独立于顶层操作系统、CPU 和系统体系结构。VDBE 的字节代码（称为虚拟机语言）由 100 多个被称为操作码（opcodes）的可能的任务构成，所有这些操作都是围绕数据库进行的。VDBE 是一个专为数据处理设计的虚拟机。它的指令集中所有的指令或者用来完成具体的数据库操作（比如打开一个表的游标、做记录、提取一列或者开始一个事务等），或者以某种方式控制栈为完成这些操作做准备。这些命令在一起并以恰当的顺序组合，就可以满足复杂 SQL 命令的要求。SQLite 中的所有 SQL 语句——从选择和更新记录到创建表、视图以及索引——都是首先编译成虚拟机语言，形成一个独立的定义了如何完成给定的命令的指令集。例如，以下面的语句为例：

```
SELECT name FROM episodes LIMIT 10;
```

SQLite 会显示编译后的 VDBE 汇编程序，如表 1-1 所示。

表 1-1　VDBE 汇编程序

addr	opcode	p1	p2	p3	p4	p5	comment
0	Trace	0	0	0		00	
1	Integer	10	1	0		00	
2	Goto	0	11	0		00	
3	OpenRead	0	2	0	3	00	
4	Rewind	0	9	0		00	
5	Column	0	2	2		00	
6	ResultRow	2	1	0		00	
7	IfZero	1	9	-1		00	
8	Next	0	5	0		01	
9	Close	0	0	0		00	
10	Halt	0	0	0		00	
11	Transactio	0	0	0		00	
12	VerifyCook	0	4	0		00	
13	TableLock	0	2	0	episodes	00	
14	Goto	0	3	0		00	

程序由 15 条指令组成，这些指令以特定的顺序和操作数执行，返回表 episodes 的前 10 个记录的 name 字段（episodes 表是本书示例数据库的一部分）。

从多个方面都可以看出，VDBE 是 SQLite 的核心。它之前的所有模块都是用于

创建 VDBE 程序的，它之后的所有模块都是用于执行 VDBE 程序的，每次执行一条指令。

后端

后端由 B-tree、页缓存（page cache）以及操作系统接口组成。B-tree 和 pager 一起作为信息代理。它们使用的货币就是数据库页，这些页是具有相同大小的数据块，就像用于运输的集装箱。页里面的“货物”是表示信息的大量位（bit），这些信息包括记录、字段和索引项等。B-tree 和 pager 都不知道信息内容，它们只负责移动和排列这些页，并不关心其中装的是什么。

B-tree 的职责就是排序。它维护着多个页之间错综复杂的关系，这些关系能保证快速定位并找到一切有联系的数据。B-tree 将页面组织成树状结构（这也是它叫做 B-tree 的原因），这种组织结构很适合搜索，页面就是树的叶子。pager（SQLite 的一种数据结构）帮助 B-tree 管理页面，它负责传输。pager 根据 B-tree 的请求从磁盘读取页面或者向磁盘写入页面。磁盘操作是目前计算机必须做的工作中最慢的事情之一。因此，pager 试图通过将频繁使用的页面缓存在内存中来加速这一操作，从而最小化与硬盘直接交互所花费的时间。这种树是为查询而高度优化了的。它使用特殊的算法来预测将来要使用哪些页，从而让 B-tree 保存该页面以便尽可能快地工作。pager 的功能描述也包含事务管理、数据库锁以及崩溃恢复，其中许多功能是通过 OS 接口实现的。

像文件锁一样的很多事情在不同的操作系统上实现是不同的。OS 接口为 SQLite 其他模块提供了屏蔽这些差异的抽象层。最终的结果就是其他模块看到的是一个一致的做这些锁文件一类事情的接口。所以，pager 不用担心在 Windows 上以一种方式锁文件，而在其他不同操作系统上（例如 UNIX）上使用另一种方式。SQLite 让 OS 接口负责这些，它只需要对 OS 接口说“锁定这个文件”，OS 接口会根据正在运行的系统指出如何做。OS 接口不仅保证了其他模块代码的简单整洁，也将那些凌乱的问题在一个地方集中管理起来。这使得 SQLite 很容易移植到不同的操作系统上——所有的必须解决的 OS 问题已经清晰标示出来，并在 OS 接口 API 中以文档方式记录了。

8

工具和测试代码

工具模块中包含各种各样实用的功能和诸如内存分配、字符串比较、Unicode 转换

之类的公共服务。这个模块基本上是一个包罗万象的工具箱，很多其他模块都需要调用和共享它。

测试模块中包含了无数的回归测试用例，用来检查数据库代码的每个细微角落。这个模块也是使 SQLite 如此可靠的原因之一：它执行了大量的回归测试，任何人都可以运行和改进这些测试。

SQLite 的特性和设计理念

尽管 SQLite 非常小，但却具有如此之多的特色和功能。它支持 ANSI SQL92 标准的大子集（包括事务、视图、检查约束、外键、关联子查询和组合查询等），以及其他很多关系型数据库的特性，如触发器、索引、自动增长字段和 LIMIT/OFFSET 子句。SQLite 还有很多独特的特色，如内存数据库、动态类型和冲突解决——其他数据库中称为 merge 或 upsert，后面会详细解释。

正如本章开始时所提到的，SQLite 有很多指导原则或特性，这些原则或特性或多或少为定义它的设计理念和实现而服务。下面就来详述这些原则。

零配置

SQLite 设计之初，就明确了不需要 DBA。配置和管理 SQLite 就像得到它一样容易。SQLite 包含了足够的正好适合于程序员头脑的特色，像它的程序库一样，只需要较少的内存就可以运行。

移植性

SQLite 在设计时特别注意移植性。它可以编译运行在 Windows、Linux、BSD、Mac OS X 及商用 UNIX 系统如 Solaris、HPUX 和 AIX 上，还可以应用于很多嵌入式平台如 QNX、VxWorks、Symbian、Palm OS 和 Windows CE。它可以无缝地工作在 32 位和 64 位体系结构中，并能同时适应大字节序和小字节序。SQLite 的移植性并不只限于软件上，还表现在其数据库文件上。SQLite 的数据库文件在其所支持的所有操作系统、硬件体系结构和字节顺序上都是兼容的二进制。您可以在 Linux 工作站上创建一个 SQLite 数据库，然后在 Mac 或 Windows 的机器上或者 iPhone 和其他设备上使用它，而不需要做任何转

换和修改。此外，SQLite 数据库可以支撑高达 2TB 的数据量（受操作系统上最大文件大小的限制），还内置了同时支持 UTF-8 和 UTF-16 编码。

紧凑性

SQLite 的设计是轻量级的，只包含 1 个头文件、1 个库，以及关系型的不需要外部数据库的服务器。所有的这些文件都被打包在 0.5MB 之内，比许多您访问的 Web 页面还要小。

SQLite 数据库文件是普通的操作系统文件。与操作系统无关，SQLite 数据库中的所有对象——表、触发器、模式、索引以及视图——都包含在一个操作系统文件中。SQLite 使用可变长度的记录，只分配数据存储每个字段的最小量。在 varchar（100）列中，如果放入 2 字节的字段，则只占 3 字节的空间（额外的字节用于记录它的类型信息），而不是 100。

简单性

作为程序库，SQLite 的 API 可以算是最简单、最易用的 API 之一。SQLite API 既有很好的文档，又很直观。它的设计有助于您从多方面定制 SQLite，例如用 C 语言实现您自己定制的 SQL 函数。更好的是，开源社区创建了很多语言和程序库用来与 SQLite 交互，具体包括 Perl、Python、Ruby、Tcl/Tk、Java、PHP、Visual Basic、ODBC、Delphi、C#、VB.NET、Smalltalk、Ada、Objective C、Eiffel、Rexx、Lisp、Scheme、Lua、Pike、Objective Camel、Qt、WxWindows、REALBASIC 等语言的扩展。可以从 SQLite Wiki：www.sqlite.org/cvstrac/wiki?p=SqliteWrappers 中找到一个全面的列表。

SQLite 的模块化设计包含很多创新的想法，这使得 SQLite 具备全功能，又是可扩展的，同时通过基础代码使整个代码在很大程度上保持了简单性。每一个模块都是专门的、独立执行具体任务的子系统。这种模块化的特性使得独立开发每个子系统变得更容易，即从一个模块向另一个模块传递——从编译和计划到执行和实现调试查询。最终结果就是在前端（SQL 编译器）和后端（存储系统）中间有一个清晰定义的明确的界限，这使得两者能够独立编码。这种设计可以更容易向数据库引擎添加新功能，能更快地调试，使得最终整体的可靠性更好。

灵活性

SQLite 的几个特性使其成为一个非常灵活的数据库。作为一款嵌入式数据库，SQLite 在以下两方面都做得最好：强大而灵活的关系型数据库前端，简单而紧凑的 B-tree 后端。有了这些，不用配置大型数据库服务器，不用担心网络或者连接问题，没有平台限制，不用付许可证费或者版税。相反，您将其放到应用程序中就可以获得简单的 SQL 支持。

自由授权

SQLite 的全部代码都在公共域中，并且没有许可证。SQLite 的任何一部分都没有附加版权要求。所有曾经为 SQLite 项目贡献过代码的人都签署过一个宣誓书明确放弃对他们所贡献代码的任何版权。因此，以任何形式使用 SQLite 的代码都不会有法律方面的限制。您可以修改、合并、发布、出售或将这些代码用于任何目的——商业或者非商业的，不需要支付任何费用，也没有任何限制。

可靠性

SQLite 源代码不但免费，而且还写得很优秀。SQLite 源代码包含大约 70,000 行标准 ANSI C 代码，代码模块清晰，注释完整，并且易理解、易定制，方便获取。对一个合格的 C 程序员而言，只要有足够的时间去学习，理解 SQLite 的任何一部分或者整个代码完全是在能力范围内的。

此外，SQLite 代码也提供一个全功能的 API，通过添加用户自定义的函数、聚集、排列规则以及支持一些安全操作，可以定制和扩展 SQLite。

虽然 SQLite 的模块化设计对它的整体可靠性贡献很大，但是它的源代码还是经过完整测试的。SQLite 的核心软件（库和工具）由约 70,000 行代码组成，但发布的程序中还包含有超过 45,000,000 行的测试代码，2009 年 7 月测试代码覆盖了 100%的核心代码。您可以询问任何开发者，要做到如此全面测试重要的代码有多艰难，同时您也知道为什么人们对 SQLite 的可靠性如此有信心。

易用性

SQLite 还具备一些独特的功能，这些功能从很大程度上提高了其易用性，包括动态

类型、冲突解决和可以将多个数据库“附着”到一个连接上。

SQLite 的动态类型与脚本语言中的一些东西类似（例如，Ruby 中的“duck typing”）。具体地讲，变量的类型是由它的值决定的，而不是静态类型语言中使用的声明决定的。绝大部分数据库系统会限制字段中的值只能是各自列中声明的类型。例如，整数列中的字段只能是整数或 NULL。在 SQLite 中，列可以有声明的类型，但字段可以随便偏离它们，就如脚本语言中的变量可以重新赋予不同类型的值一样。这对于原型设计是特别有用的，因为 SQLite 不会强迫您明确地改变列的类型，您只需要改变程序如何存储列中的信息，而不是持续更新模式和重新加载数据。

冲突解决是另一个独特的功能，它使得编写 SQL 更容易。这一功能内建在许多 SQL 操作中，可以用来执行称为“懒惰更新”的操作。假设有一个记录需要插入，但是您不确定数据库中是否已经存在一个了。可以不用先写 SELECT 语句去查询，如果存在，就将您的 INSERT 语句重新改写成 UPDATE 语句，冲突解决可以帮助您。“试图插入该记录，如果发现相同主键的记录，就更新该记录”。现在就可以将必须写三个不同的 SQL 语句（例如，SELECT、INSERT，以及可能的 UPDATE）转换为一个 INSERT ON REPLACE（…）语句。其他关系型数据库使用 UPSERT 或 MERGE 语句在模仿这种冲突解决方法，但是 SQLite 已经抢占先机。您可以将冲突解决构建到表的定义中，不需要在将来的 INSERT 语句中再次指明。事实上，您也可以免除在该表上写 UPDATE 语句——只需要使用 INSERT 语句让 SQLite 自己使用模式中定义的冲突解决去识别需要做什么。

最后，SQLite 可以将外部数据库“附着”到当前的连接中。假如您当前连接到一个数据库 foo.db，同时需要另外一个数据库 bar.db 工作。不用打开一个单独的连接，然后在它们之间来回切换，可以简单地将感兴趣的数据库用下面一个 SQL 语句附着到当前连接：

```
ATTACH database bar.db as bar;
```

bar.db 中所有的表现在都可以访问了，就好像它们存在于 foo.db 中一样，完成时也可以剥离。这使得在数据库之间的各种操作如同复制表一样容易。 11

性能和限制

SQLite 是一个很快的数据库，但“快”这个词本身是一个主观的和模糊不清的词。

坦白地讲，对于有些事情，SQLite 比其他数据库做得快，但有些事情比不上其他数据库。利用 SQLite 提供的配置参数，SQLite 是足够快速和高效的。与大多数数据库一样，SQLite 使用 B-tree 做索引，使用 B+-tree 处理表。因此，在对单表进行查询时，平均而言，SQLite 与其他数据库一样快（至少不慢于）。简单的 SELECT、INSERT 和 UPDATE 语句是相当快速的——几乎与内存（如果是内存数据库）或者磁盘同速。SQLite 通常要快于其他数据库，因为它在处理一个事务开始，或者一个查询计划的产生方面开销较小，并且没有调用服务器的网络或认证以及权限协商的开销。它的简单性使它更快。

但是随着查询变大变复杂，查询时间使得网络调用或者事务处理开销相形见绌，SQLite 将会与其他数据库一样，这时一些大型的设计复杂的数据库开始发挥作用了。虽然 SQLite 也能处理复杂的查询，但是它没有精密的优化器或者查询计划器。SQLite 知道如何使用索引，但是它没有保存详细的表统计信息。假如执行 17 路 join，SQLite 也会连接表并给您结果，并不像您在 Oracle 或者 PostgreSQL 中期望的那样，SQLite 没有通过计算各种替代查询计划并选择最快的候选计划来尝试判断并优化路径。因此，假如您在大型数据集合上运行复杂的查询，SQLite 与那些有复杂设计的查询计划器的数据库运行一样快的机会是非常小的。

一些情况下，SQLite 可能不如大型数据库快，但大多数的这些情况是预期的。SQLite 是为中小规模的应用程序设计的一个嵌入式的数据库，这些限制是设计目的预见到的。许多新用户错误地认为使用 SQLite 可以代替大型关系数据库。事实是有时可以这样做，有时不行，这完全取决于您准备用 SQLite 来做什么。

一般情况下，SQLite 的局限性主要有以下两方面：

- **并发**。SQLite 的锁机制是粗粒度的，它允许多个读，但是一次只允许一个写。写锁会在写期间排他地锁定数据库，其他人在此期间不能访问数据库。SQLite 已经采取措施以最小化排它锁所占用的时间。通常情况下，SQLite 中的锁只保持几毫秒。但是按照一般经验，如果您的应用程序有很高的写并发（许多连接竞争向同一数据库写），并且是时间关键型，您可能需要其他数据库。需要实际测试您的应用程序才能知道能获得怎样的性能。作者曾见过一个简单的网络应用程序中，SQLite 用 100 个并发连接每秒处理 500 多个事务。您的事务所修改的记录数以及查询所涉及的复杂性可能有所不同。什么样的并发性是可接受的，这取决于具体的应用程序，只能通过直接测试来判断。总之，对任何数据库都是真理：直到您做了实际测试才知道应用程序能获得怎样的性能。

- **网络**。虽然 SQLite 数据库可以通过网络文件系统共享，但是与这种文件系统相关的潜在延时会导致性能受损。更糟的是，网络文件系统实现中的一些缺陷也使得打开和修改远程文件——SQLite 或其他——很容易出错。如果文件系统的锁实现不当，可能允许两个客户端同时修改同一个数据库文件，这样必然会导致数据库出错。实际上，并不是因为 SQLite 的实现导致 SQLite 不能在网络文件系统上工作。相反，SQLite 在底层文件系统和有线协议下工作，但是这些技术有时并不完善。例如，许多 NFS 使用有缺陷的 fcntl()，这意味着锁可能并不像设想的那样工作。一些较新版本的 NFS，如 Solaris NFSV4 就可以工作正常，SQLite 需要的锁机制也是相当可靠的。然而，SQLite 开发者既没有时间，也没有资源去验证任何给定的网络文件系统在所有的情况下都可以无缺陷工作。

再次申明，绝大部分限制是有意设计的——它们是 SQLite 设计的结果。例如，支持高度的写并发会带来很大的复杂性，这将使 SQLite 的简单性无法保持。同样，作为嵌入式数据库，SQLite 是有意不支持网络的，这一点并不奇怪。总之，SQLite 不能做的都是它所能做的直接结果。它开始的设计就是一款模块化、简单、紧凑的以及易于使用的嵌入式关系数据库，它的基础代码都在使用它的程序员的掌握范围内。从多方面看，它能完成许多其他数据库不能做的事情，例如，运行在电力消耗实际上是一项制约因素的嵌入式环境中。

尽管 SQLite 的实现已经相当好了，但仍有部分特性未能实现，这些特性有：

- **完整的触发器支持**。SQLite 支持几乎所有的标准触发器功能，包括递归触发器和 INSTEAD OF 触发器。但是对于所有的触发器类型，当受触发器查询影响的每一行做评估时，SQLite 需要 FOR EACH ROW 行为。ANSI SQL92 也说明了当前不支持 FOR EACH STATEMENT。
- **完整的修改表结构支持**。目前只支持 RENAME TABLE 和 ADD COLUMN 类型的 ALTER TABLE 命令。其他类型的 ALTER TABLE 操作，例如 DROP COLUMN、ALTER COLUMN 以及 ADD CONSTRAINT 还未实现。
- **右外连接与全外连接**。左外连接（LEFT OUT JOIN）已经支持，但是右外连接（RIGHT OUTER JOIN）和全外连接（FULL OUTER JOIN）还未实现。所有的右外连接在语义上都有相同的左外连接，相反也成立。通过简单逆向表的顺序和修改连接限制，左外连接可以作为右外连接的实现。全外连接可以通过组合左外连

接和 UNION 以及在 WHERE 子句中进行恰当的 NULL 过滤实现。

- **可更新的视图。**SQLite 的视图是只读的，您不能在视图上执行 DELETE、INSERT 或者 UPDATE 语句。但是您可以创建一个启动对视图进行 DELETE、INSERT 或者 UPDATE 的触发器，在触发器内完成您需要执行的操作。

- **窗口功能。**ANSI SQL 99 的新功能之一就是窗口功能。该功能提供结果集的后处理分析，例如排序、平均移动以及超前和滞后计算等。SQLite 目前支持 ANSI SQL 92 的一部分，因此，它不支持像 RANK()、ROW_NUMBER()等。
- **授权和撤销。**由于 SQLite 能读写普通的磁盘文件，因此，唯一可以应用的访问权限就是所在操作系统的普通文件的访问权限。授权（GRANT）和撤销（REVOKE）命令一般是在高端系统中使用的，这些系统中有多个用户，不同用户对数据库中的数据有不同的访问级别。SQLite 模型中，应用程序是主用户，能够访问整个数据库。这种模型中的访问明确定义为应用程序级——具体地说，就是应用程序可以访问数据库文件。

除了这里列出的，SQLite Wiki 上有个专门的页面列出了 SQLite 不支持的 SQL。网址是 www.sqlite.org/cvstrac/wiki?p=UnsupportedSql。

本书面向的读者

SQLite 不仅用途广泛，同时应用众多，因此，也有很多读者。不管您是程序员、网络开发者或者系统管理员，或者仅仅是随意翻看准备学习关系数据库的用户，本书都有助于您理解并掌握 SQLite 的使用。

如果您刚接触关系型数据库，那么 SQLite 是一个不错的开始入门的数据库。本书不仅帮助您从 SQLite 入门，而且会使你逐步成为 SQL 合格的使用者。SQLite 将为您提供一个良好的基础，这样您可以转移到大型关系型系统，进一步扩展高级功能。

对于程序员而言，本书只是假设您已了解与 SQLite 相关的编程语言和关系理论的基本知识。此外，本书并不是 API 的解释文档。它与 API 文档是有区别的，API 文档只是说明接口如何工作。对于任何数据库而言，必须对数据库内部是如何工作的有所了解，这样才能掌握它。每种数据库都有它独特的架构，具体的关系型特性和一些重要的限制，一个好的程序员应该了解这些，并在写代码时考虑这些因素。SQLite 虽然简单明了，但

是也不例外。作为程序员，需要知道一些关于内部是如何处理数据的，这样才能使 SQLite 与应用程序工作得更好。本书将告诉您 SQLite 是如何处理数据的。本书覆盖了 API 并探究了它们如何在 SQLite 体系架构中工作，使 C 语言程序员、Web 开发者以及脚本写作者能写出更好、更有效的代码。这不仅使您能更好地理解 SQLite 能做什么，也知道它不能做什么。架构知识比起那些规则来说，它能使您更好地理解 SQLite 什么时候适合您的项目，什么时候不适合。您将知道是否、何时，以及哪些项目需要考虑其他方法。

本书的重要目的之一是：通过方法传递概念——适当解决如何以及为什么的问题。读完本书后，您将从概念和理论上理解它们是如何工作的。最后，本书写作中穿插了很多图片和例子，它们用来说明其中的理论主题并提供实际案例。

本书的组织结构

本书划分为三部分：SQLite 数据库、SQLite 编程库、参考资料。第 2～4 章讨论 SQLite 数据库的内容，第 5～8 章讨论 SQLite 编程库。各章简介如下：

第 1 章，“SQLite 介绍”，介绍了 SQLite 的主要功能、起源和历史，以及本书的范围和目标。

第 2 章，“入门”，涵盖了如何获取和使用 SQLite。说明了如何获取二进制形式和源代码形式的 SQLite，以及如何在各种平台上编译和构建。解释了如何使用 SQLite 的命令行工具来创建和使用数据库。

第 3 章，“SQLite 中的 SQL”，提供了 SQL 的一些背景知识。介绍了许多基本的 SQL 命令，例如创建表和选择数据的命令。本章既对 SQLite 的 SQL 做了全面综述，又对 SQL 自身进行了快速介绍。

第 4 章，“SQLite 中的高级 SQL”，继续第 3 章的内容，覆盖了 SQLite 中 SQL 实现的复杂的查询。

第 5 章，“SQLite 设计与概念”，罗列与 SQLite 一起编程的基础。说明了 SQLite API、结构以及两者是如何关联的。引入了诸如事务和锁的话题。向程序员解释了 SQLite 内部是如何工作的，以及编程使用 SQLite 时需要记住哪些东西。

第 6 章，“核心 C API”，涵盖了部分与执行查询相关的 SQLite C API。从连接数据库、执行查询、获取数据、管理事务到追踪代码。该章包含了所有与查询和数据处理相

关的 API。

第 7 章，“扩展 C API”，涵盖了剩余部分的用于定制和扩展 SQLite 的 C API。SQLite 提供了实现用户定义 SQL 函数、聚合以及排序规则的工具。该章说明如何实现这些功能并提供一个如何使用它们的实例。

第 8 章，“SQLite 语言绑定”，提供一些流行语言的 SQLite 编程简明介绍，如 Perl、Python、Ruby、Java 和 PHP。

第 9 章，“苹果 iPhone 和 iPad 上的 SQLite”，关注在苹果移动平台上开发时如何使用 SQLite。

第 10 章，“谷歌 Android 设备上的 SQLite”，覆盖了 SQLite 如何支持 Android 平台，以及如何在这些设备上做 SQLite 相关的开发。

第 11 章，“SQLite 内核”，探究了 SQLite 的内部工作机制。对源代码进行高级概述，简要概述了主要的子系统是如何实现的。为程序员和想在 SQLite 上做工作的开发者提供了 SQLite 设计决策、假定以及取舍的深度理解。

最后，附录部分包含 SQLite C API 和 SQL 语法的完整参考指南。

示例数据库

本书附录的示例数据库可以从 Apress 网站（www.apress.com）下载。所有的数据库都是 SQL 格式的，可以按照本书第 2 章的步骤使用 SQLite 的命令行程序创建数据库。随着本书的深入，会进一步介绍和探究示例数据库。

所有的源代码都可以在线获得。所有的例子都可以在 Windows 和 Linux、UNIX 平台上编译并运行。例如，makefile 既包含了 Linux、UNIX 环境，又包含为 Windows 用户创建的 Visual C++项目。

附加说明

SQLite 网站上有大量的信息，包括官方文档、邮件列表、Wiki 以及其他常用信息，其网址是 www.sqlite.org。SQLite 社区也非常不错，几乎能在 SQLite 邮件列表中找到一切需要的东西。此外，SQLite 作者提供 SQLite 专业培训和支持，包括定制程序（移植到嵌入式平台等）。包含本地加密和为嵌入式应用做较小优化的增强版的 SQLite。更多信息可登录 www.hwaci.com/sw/sqlite/prosupport.html。

总结

SQLite 与其他大型数据库如 Oracle 或者 PostgreSQL 具有明显的区别。虽然它们都是关系型数据库，但是 Oracle 或者 PostgreSQL 就如同电子巨头，而 SQLite 就如同数字化的瑞士军刀。相反，大型关系数据库是为数以千计的用户设计的，而 SQLite 是为数以千计的应用设计的。SQLite 不只是一个数据库，虽然它本身是一个工具，但同时它也是制造工具的工具。它能使开发者、用户或者管理员快速方便地将不相关的数据排好序，并以最小的代价按照需要控制它们。

SQLite 是公开、免费的软件。您可以用它或者它的源代码做任何事情，并且没有许可证、安装程序和其他限制，只需要复制它，就可运行。SQLite 也是可移植的、已测试过的、可信赖的。它有着清晰的模块化设计，这使得它的系统简单、易于开发，便于调试。除了优秀的设计外，它还经过良好的测试。已经有很多代码测试 SQLite，这比 SQLite 中包含的测试代码还多。毫无疑问，过去 5 年的历史已经证明 SQLite 是稳定可靠的、可信赖的数据库。

16

最后，SQLite 也是有趣的，至少作者是这样认为的。希望您同样能发现 SQLite 是有用的并享受它。

如果您对本书或者其中的例子有任何意见或建议，请给作者发邮件，地址是 sqlitebook@gmail.com (Michael) 或 grantondata@gmail.com (Grant)。

第2章

入　门

无论您使用何种操作系统，SQLite 都很容易上手。对大多数用户而言，不需要任何经验，5 分钟内就可学会安装 SQLite 并创建一个新的数据库。本章内容包括安装 SQLite 的所有细节和如何使用 SQLite 管理数据库。完成本章的学习后，您将知道从哪里获取 SQLite 软件或者源代码，以及如何在各种平台上安装或编译 SQLite。您将可以使用 SQLite 在数据库中进行创建表、视图、索引以及查询、备份和恢复等相应的操作。

此外，您将学习到管理 SQLite 数据库所需的一切知识，包括如何创建、查看以及检查这些内容。最后，介绍几个工具，它们可以和 SQLite 在各种环境中一起工作。本章使用 SQL 描述一些 SQLite 命令行程序的例子。即使对 SQL 不熟悉，您依然能够理解这些例子。本书第 3、4 章会详细介绍 SQL。

何处获取 SQLite

SQLite 的官方网站（www.sqlite.org）既提供预编译好的二进制版本的 SQLite，也提供源代码。二进制版本的 SQLite 可在常见的平台上使用，例如苹果的 Mac OS X、Windows 和 Linux。

根据不同的使用方式，有以下几种形式的二进制程序包可供选择。

- **sqlite3 命令行程序（CLP）**：该版本的 SQLite 命令行程序通过静态链接将数据库引擎编译在内，是自我包含的独立的程序。它提供了一种便捷的通过命令行与 SQLite 数据库一起工作的方式，用户不用关心 SQLite 共享库是否已经安装在系统上或安装在正确的位置。如果是较新版的 Linux 或者苹果 OS X 系统，几乎可以肯定已经安装了 sqlite。
- **SQLite 共享库（DLL 或者“so”）**：共享库就是将 SQLite 数据库引擎打包进共享

库（so），或者 Windows 动态链接库（DLL）。动态链接 SQLite 的程序使用该库。这种形式使 SQLite 升级更容易，且无须重新编译依赖 SQLite 的软件。注意，目前没有为苹果 OS X 提供编译的.dylib 共享库。

- **SQLite Analyzer**：分析工具可以用来检查给定 SQLite 数据库文件的特性和统计信息，例如，文件中的物理数据分布和布局、相对碎片、内部数据大小和空闲空间以及其他内容。这些信息对于性能优化是非常有用的。
- **Tcl 扩展**：二进制版本在 SQLite core 中包含了 Tcl 语言扩展，这样您就可以从 Tcl 语言连接到 SQLite。SQLite 团队已经提供了 Tcl 和 C 这两种语言的绑定，其他语言的绑定知识将在第 8 章讨论。

根据平台的不同，SQLite 提供了两种源代码形式。不同源代码版本之间的主要差别是代码是否合成在一个源文件和一个头文件中。也有一些源代码包已经包含了 Tcl 扩展架构（TEA），供那些想将 SQLite 和 Tcl 绑定在一起编译的人使用。

SQLite 合并后的源代码有两种——zip 文件与 gzipped tarball 文件。这两种版本的源代码本身没有差别，一般推荐使用 zip 文件，它也是 Windows 上最适合编译的版本。zip 版本的源代码已经经过了一定的预处理，具备了 GUN Autoconf 或其他构建工具执行产生的代码。这样 Windows 用户就不需要寻找 Windows 平台上对应的这些构建工具了。Gzipped tarball 是给那些诸如 Linux、UNIX 风格、苹果 Mac OS X 和 BSD 变种以及其他系统的用户使用的。这些操作系统平台一般在发行版本中包含了类似于 GNU Autoconf 这样的用来做预处理和编译、构建工作的工具。因此，这些平台的用户可以使用这种源代码，而不需要寻找额外的编译 SQLite 的工具。

Windows 上的 SQLite

无论你是作为终端用户还是程序员，或是将 SQLite 作为学习数据库关系理论和 SQL 的工具，SQLite 都可以很容易地安装在 Windows 上。本节我们将讨论所有相关的内容——安装可用的二进制包、用最流行的编译器从源代码构建所有的内容。先易后难，我们从简单的开始，然后介绍有技术挑战的内容。

获得命令行程序

SQLite 命令行程序（后面简称为 CLP）是目前开始使用 SQLite 的最好选择。按照

如下步骤获取 CLP：

1. 打开浏览器进入 SQLite 主页：www.sqlite.org。
2. 单击页面顶部的下载链接，进入下载页面。
3. 滚动鼠标到 Precompiled Binaries For Windows，下面有个名为 sqlite-3_x_y_z.zip 的文件，其中 x、y 和 z 代表小版本号。旁边应该还有个注释："访问并修改 SQLite 数据库的命令行程序"（A command-line program for accessing and modifying SQLite databases）。下载该文件到 Windows 环境下。 19
4. 使用 Windows 自带的解压工具或者第三方压缩工具，例如 WinZip 或 7-Zip，解压该文件。zip 文件应该包含了一个名为 sqlite3.exe 的文件，可以从解压文件所在位置开始使用 sqlite，如果您想从任何目录运行 CLP，需要将该文件复制到 Windows 系统路径下。默认的应该可以在所有版本的 Windows 中工作的路径是根分区下的\windows\system32（对大部分系统而言，C:\就是根分区）。

注意：如果您不知道 Windows 系统路径，可以从命令行执行如下命令：

```
Echo %PATH%
```

命令执行结果返回一串通过分号隔开的目录字符串——这就是系统路径。也可以通过图形化方式从控制面板获得同样的信息，单击开始→控制面板，选择系统图标，在结果对话框中选择高级标签页，单击环境变量按钮。在系统变量类表下半部中双击 Path 项。编辑系统变量对话框将打开，在变量值中可以看见 Path 列表。也可以给 path 添加额外的目录，只需要在目录中加上分号和新的路径就行。

5. 打开命令行窗口。可以通过不同的方式来完成，使用 Windows "开始" 菜单，选择开始→运行，在打开的下拉框中输入 cmd，单击 OK（图 2-1），即可打开 Windows 命令行窗口。如果这样行不通，试着通过 "开始→所有程序→附件→命令行" 打开。
6. 在命令行窗口输入 sqlite3 并按回车键，将出现 SQLite 命令行提示符。（如果出错，是因为没有将可执行的 sqlite3.exe 复制到系统路径目录中。重新检查您的路径，将 sqlite exe 复制到启动它的位置。）当 SQLite 命令行提示符出现时，输入.help，将出现一列与图 2-2 相似的相关命令说明。输入.exit 后退出程序。现 20

在您已经有一份可以工作的安装在系统上的 SQLite CLP 副本了。

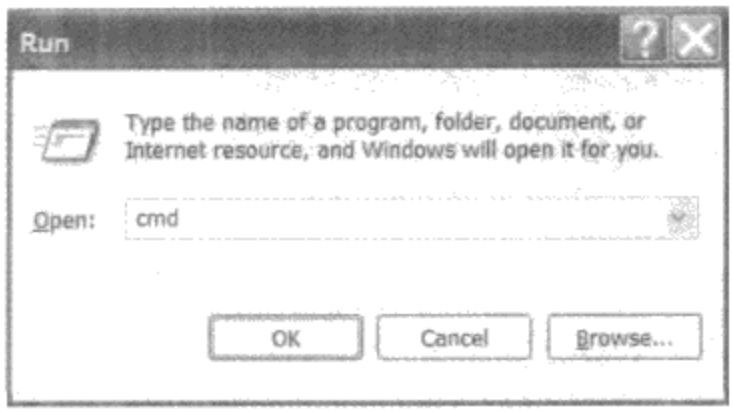


图 2-1　打开 Windows 命令行

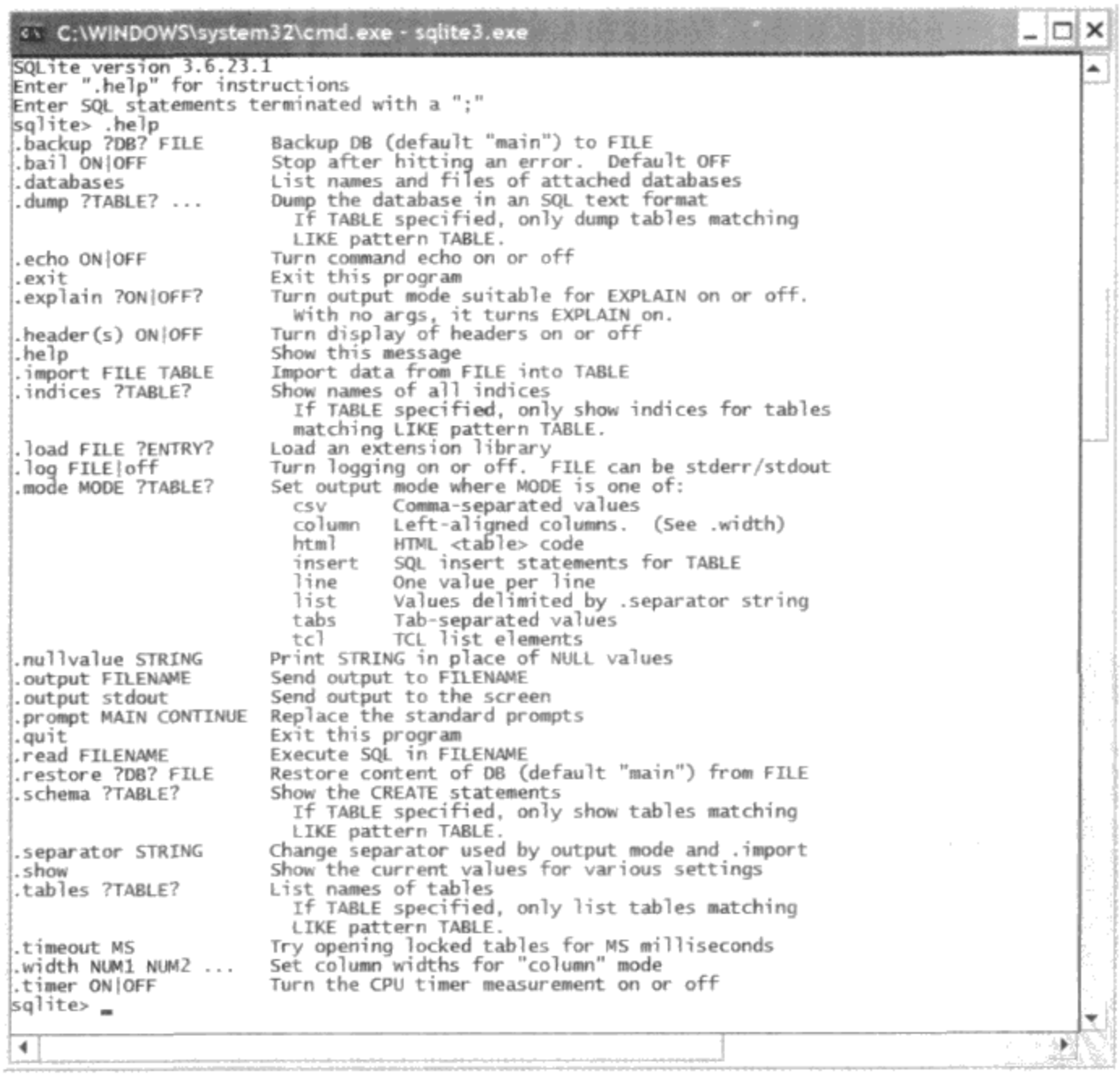


图 2-2　Windows 上的 SQLite 命令行

如果你此刻就想使用 SQLite，可以跳到“Shell 模式的 CLP”一节，下面几节针对那些想用 SQLite 写程序的开发者。 21

获取 SQLite 动态链接库（DLL)

SQLite 的 DLL 文件使得经过编译的软件可以动态链接到 SQLite。这意味着在需要 SQLite 功能时，应用程序会加载 DLL，而不是将 SQLite 代码嵌入在应用程序中。以这种方式使用 SQLite 的软件通常包含一份 SQLite DLL 副本随软件自动安装。

如果您准备用 SQLite 编程，使用 DLL 可能是最容易上手的方式。可以通过如下方式获得 SQLite DLL：

1. 进入 SQLite 主页 www.sqlite.com，在页面顶部单击下载链接，进入下载页面。
2. 在下载页面找到 Precompiled Binaries For Windows。
3. 找出 DLL zip 文件。该文件有这样的描述：This is a DLL of the SQLite library without the TCL bindings。文件名是 sqlitedll-3_x_y_z.zip 的形式，其中 x、y 和 z 是小版本号。如果您需要包含 Tcl 支持，则选择文件名为 tclsqlitedll-3_x_y_z.zip 的文件。
4. 下载并解压该文件。解压后的内容应该包括两个文件：实际的 DLL 文件（sqlite3.dll）和 sqlite3.def。这里提供的 SQLite DLL 是线程安全的，也就是说，它是定义了 THREADSAFE 预处理符后编译的。这样就可以在多线程程序中使用 SQLite；也可以在多线程程序中使用该 DLL，安全地在多线程中执行同步操作。

要使用 DLL，该 DLL 需要与程序处于同一文件夹下或者在系统路径中（参考前面小节中关于 Windows 系统路径的注释），或者遵循 Windows 动态链接库加载时寻找 DLL 的规则。

如果想通过写程序使用 SQLite DLL，需要创建链接程序的导入库。借助前面提到的 sqlite3.def，完成这项工作非常简单。如果使用微软 Visual Studio 中的 C++，打开命令行，将目录转到 SQLite 所在处，运行如下命令：

```
LIB /DEF:sqlite3.def
```

将会有如下所示的常见库产生输出：

```
C:\sqlite>lib /DEF:sqlite3.def
Microsoft (R) Library Manager Version 10.00.30319.01
```

```
Copyright (C) Microsoft Corporation.  All rights reserved.
LINK : warning LNK4068: /MACHINE not specified; defaulting to X86
    Creating library sqlite3.lib and object sqlite3.exp
```

该命令将产生一个名为 sqlite3.lib 的导入库和名为.exp 的导出文件。如果使用的是 MinGW（参考本章后面的“用 MinGW 构建 SQLite”），运行命令：

```
dlltool --def sqlite3.def --dllname sqlite3.dll --output-lib sqlite3.Lib
```

MinGW 的 dlltool 将会创建名为 sqlite3.Lig 的导入库，借助该库可以链接程序。通过将程序链接到该导入库，它们会被加载并在 SQLite DLL 中执行。

在 Windows 上编译 SQLite 源代码

在 Windows 环境下，从源代码构建 SQLite 很简单。根据所使用的编译器不同，有几种编译 SQLite 的方法。Windows 上最常见的环境是 Microsoft Visual C++、MinGW 或者 Eclipse（Eclipse 经常使用 MinGW 来完成 C++工作），本节会介绍这些内容。关于如何使用其他编译器编译 SQLite，可参考 SQLite Wiki（www.sqlite.org/cvstrac/wiki?p=HowTo Compile）。

稳定的源代码版本

SQLite 源代码的稳定版本可以从 SQLite 网站中下载 zip 文件。最新版本可以从 SQLite 团队维护的 Fossil 分布式源代码控制系统获得。除非您很熟悉 Fossil，否则使用已发布的源代码是最容易的方式。按照如下步骤下载稳定的源代码版本：

1. 进入 SQLite 主页 www.sqlite.org，单击下载链接，进入下载页面。
2. 在下载页面找出源代码一节。
3. 第一个列出的文件是推荐使用的 zip 文件，它包含了源代码，并且适合在 Windows 上编译。该文件的名称为 sqlit-eamalgamation-3-x_y_z.zip，其中 x、y 和 z（可能还有更小的版本号）代表小版本号。

注意：下载页面中的 zip 压缩文件与其他 tarball 文件在内容上稍有不同。虽然它们包含几乎完全相同的源代码，SQLite 发布使用了一些 POSIX 构建工具（sed、awk 等）在构建中动态产生一些 C 代码。这些构建工具默认在 Windows 上是没有的。因此，为了方便在缺乏 Linux、UNIX、苹果 Mac OS X 等系统上

的基础上构建支持的 Windows 用户，Windows 压缩文件包括了所有的预处理和生成代码。这也是为什么 Windows 用户应该使用下载页面的 zip 文件，而不是 tarball 文件的原因。也可以在 Windows 上构建 tarball，但是需要一些必备的从源代码构建的构建工具，例如 Cygwin、MinGW、GnuWin2 或者类似的工具。

4. 将文件解压到选定的目录中。解压后的内容是一份完整的适合在 Windows 上运行的 SQLite 第 3 版的源代码。

匿名 Fossil 源代码控制

如果您想体验最新的特性或想参与 SQLite 开发，可以从项目 Fossil 源代码控制系统获取 SQLite 开始。Fossil 提供同样的匿名访问，就像可能会在其他源代码控制系统，像 CVS、SubVersion、Perforce、Git 等看到的那样。Fossil 允许直接通过浏览器访问最新版本的 SQLite 源代码！如果需要，您可以让本地副本维持天、小时或分钟的同步或者有提交时保持不变。因此，如果您看到了想要使用的重要的 bug 修复或功能提交，可以让 Fossil 同步并重新编译您的代码副本。Fossil 甚至包括自动同步功能，保证您使用最新的源代码。

在 Windows 环境下从 Fossil 获得 SQLite 并不容易。与 SQLite 的二进制文件一样，Fossil 发布的是单个静态链接的文件，您可以从该项目主页 www.fossil-scm.org 上下载 Fossil。下载页面包括 Windows、Mac OS X 和 Linux 系统上最新版本的 Fossil。解压缩到您的路径下（参考先前讨论的如何确定 PATH 环境变量）。接下来，创建一个新目录，或选择现有目录，以存放 Fossil 提取的源代码。

你可以在浏览器中通过 Fossil 完成大部分源代码控制工作——Fossil 充当自己的 HTTP 服务器，处理服务器端的所有任务。初次签出 SQLite 源代码的最快方法是使用刚下载的二进制 Fossil.exe 复制在线存储库。打开命令提示符窗口，并将目录更改为您选择的 SQLite Fossil 源代码控制资料库的新的或现有的目录。然后在命令提示符下启动 Fossil clone 命令，如下所示：

```
C:\sqlite\fossil.exe clone http://www.sqlite.org/src sqlite.fossil
              Bytes         Cards         Artifacts         Deltas
Send:         49            1             0                 0
Received:     1499917       32606         0                 0
Send:         10025         225           0                 0
Received:     132364        239           7                 193
```

```
...
Send:        4655         99           0                0
Received:    1698453      125          29               69
Total network traffic: 1379539 bytes sent, 17512520 bytes received
Rebuilding repository meta-data...
32605 (100%)...
project-id: 2ab58778c2967968b94284e989e43dc11791f548
server-id: 38f0c0987054889a5d2c0b4f27b370e9e8632a16
admin-user: fuzz (password is "******")
```

命令格式是 fossil <action> <fossil repository url > <your repository name>。这种情况下，action 是 clone，意思是做一份 SQLite 存储库的完整副本。SQLite 网站提供了一份 SQLite Fossil 资料库说明文档。最后，我们提供了一个由 Fossil 创建的用来包含和管理源代码副本的文件名称。这可以是任何您喜欢的满足 Windows 文件命名约定的名称。我们之所以选择 sqlite.fossil，是因为它是一个有用的名称，可以提醒我们该文件包含什么和怎么使用它。

24

■**注意**：请注意我们针对当前“head”的源代码管理树的 SQLite 的例子，这意味着您将编译最新和最大来源的代码。当然这是一把双刃剑，给您绝对最新的开发版本，但也不是“稳定”的发布版本。如果您喜欢，随意选择一个稳定的分支即可。

您现在即可启动和使用 Fossil web 接口。从命令行中只需运行下一个命令（使用您自己的目录和适当的本地存储库文件名替换本文实例中对应的内容）。

```
C:\sqlite\fossil.exe ui sqlite.fossil
```

应该可以从 Fossil 输出看到已成功启动 HTTP 服务器，并正在侦听连接。然后很快消失到浏览器后面，Fossil 将自动打开您本地的 SQLite 资源库，如图 2-3 所示。

25

现在 Fossil 已经启动并运行。从这里可以引出需要的代码，开始工作。其他大多数的 Fossil 操作一样简单。首先，从 shell 开始，为源代码创建一个目录，然后转到该目录。

```
mkdir c:\sqlite\src
cd c:\sqlite\src:
```

现在可以让 Fossil 引出当前源代码，使用 open 命令：

```
c:\sqlite\fossil.exe open c:\sqlite\sqlite.fossil
```

将看到所有 SQLite 项目的源文件在屏幕上出现。完成后，大约有 15MB 的文件已经下载，表示您已经具备一份可以工作的源代码。



图 2-3　本地 SQLite Fossil 源代码控制系统

使用微软 Visual C++构建 SQLite DLL

使用 Visual C++通过以下步骤从源代码编译构建 SQLite DLL：

1. 启动 Visual Studio。在解压的 SQLite 源程序目录中创建一个新的 DLL 项目，可以通过选择文件→新建→项目完成该操作，在项目类型（图 2-4）中选择 Visual C++项目，并选中 Win32。选择 Win32 项目模板，在位置文本框中输入包含 SQLite 源代码的文件夹名称。例如，本例中输入 C:\temp。在名称文本框中输入包含 SQLite 源代码的文件夹名称——本例中的 src。这将在已存在的 SQLite 源代码文件夹（C:\sqlite\src）中创建一个 Visual C++项目，单击 OK。
2. Win32 应用程序向导将自动打开（图 2-5）。选择应用程序设置并将应用程序类 26

型设置为 DLL。确保选择空项目，单击完成，这将会创建一个空白的 DLL 项目。

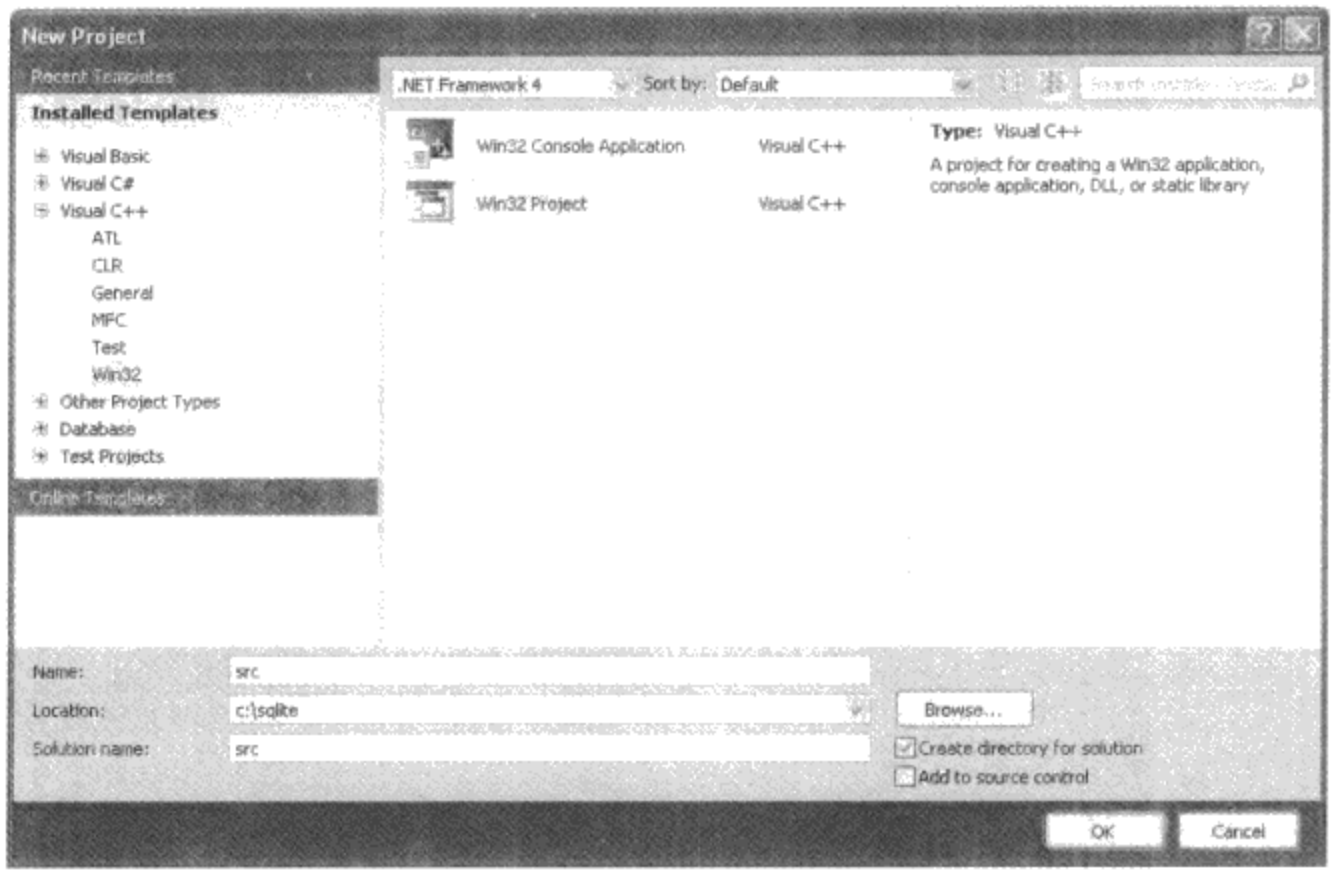


图 2-4　创建新的 Visual C++项目

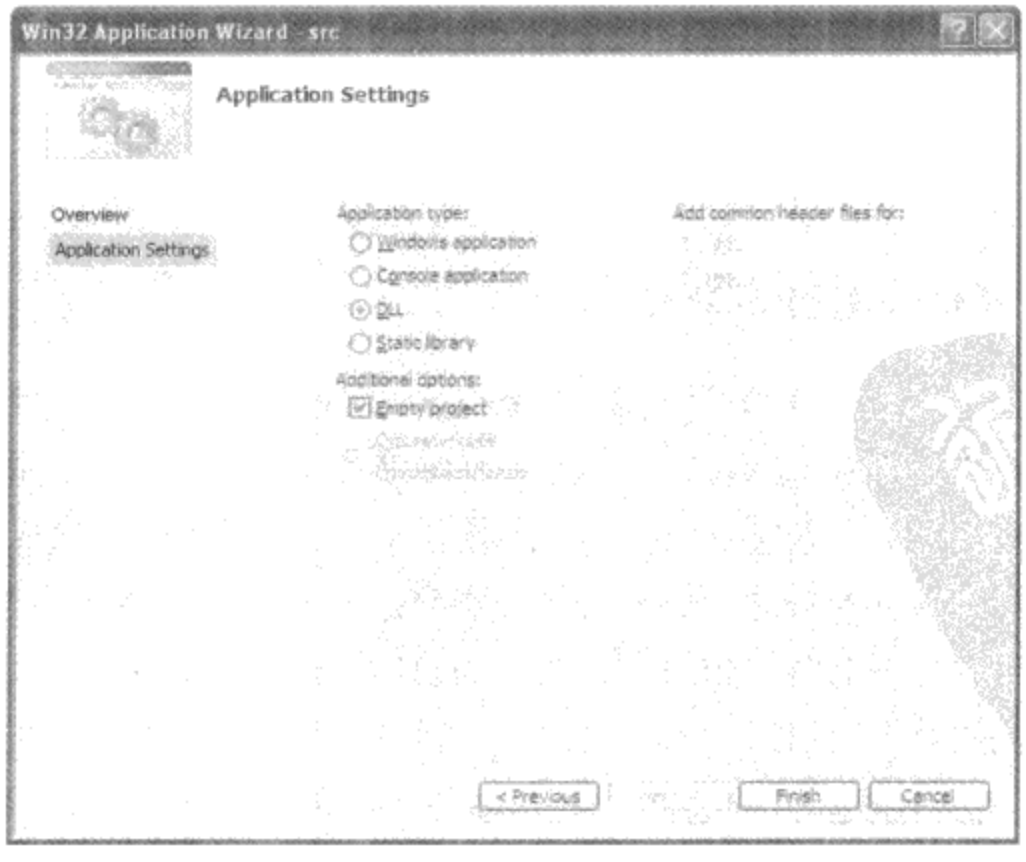


图 2-5　Win32 应用程序向导

3．将 SQLite 源文件和头文件加入项目中。选择项目→添加已存在项，将文件夹中除了 tclsqlite.c 和 shell.c 外所有的.c 文件和.h 文件添加到项目中（tclsqlite.c 是用来支持 Tcl 的；shell.c 是用来创建 SQLite CLP 的，创建 DLL 的项目不需要这两个文件）。

4．指定导出或者模块定义文件（.def）。该文件定义了向链接到该库的程序导出什么符号（函数）。SQLite 源代码文件包含了这样的文件（sqlite3.def）。在属性页面对话框的配置下拉框中选择所有的配置（图 2-6），单击连接器文件夹，单击输入子菜单。在模块定义文件属性页面输入 sqlite3.def。现在准备工作已经完成，可以构建 DLL 了。

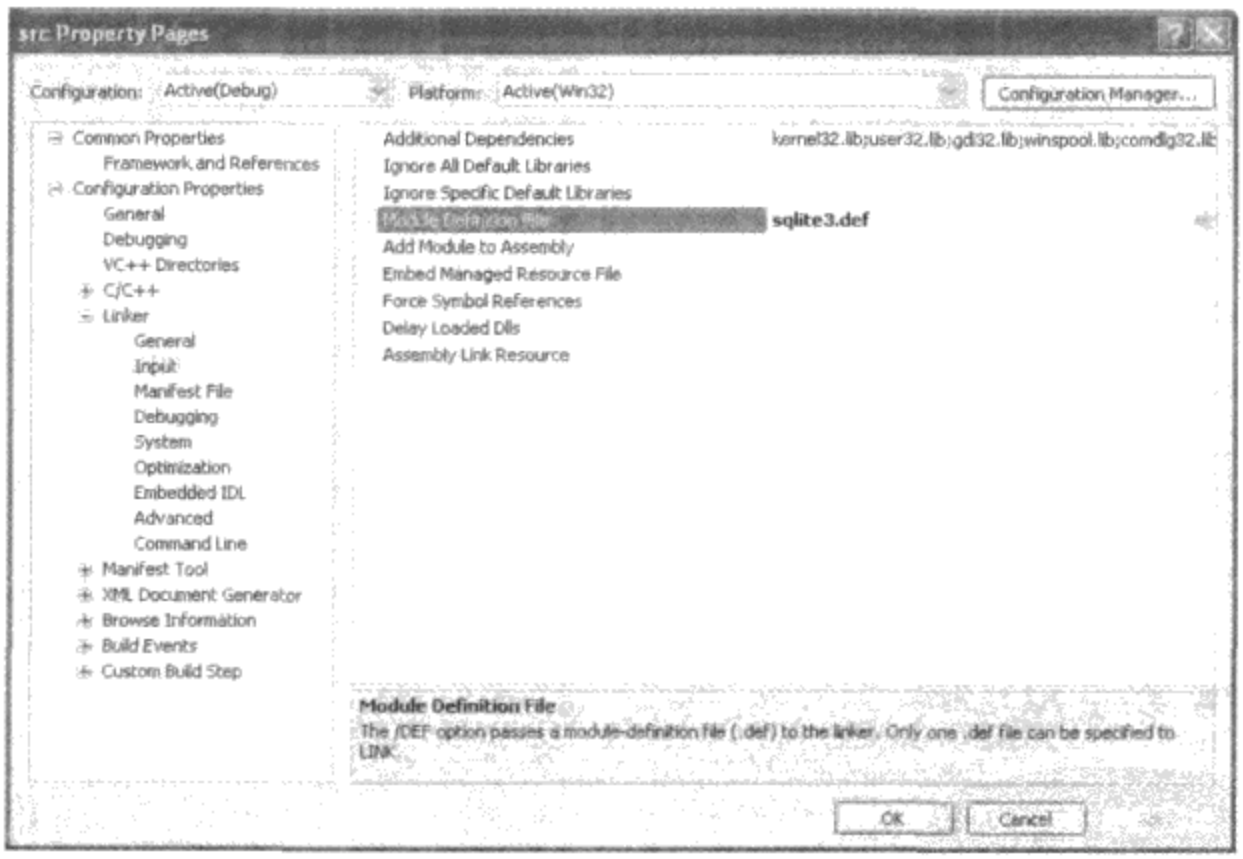


图 2-6　项目属性

5．从主菜单中选择构建→构建，开始构建 sqlite DLL。

6．一旦构建了 DLL，确保如“获得 SQLite DLL”一节中描述的那样创建导入库。

用 Visual C++构建动态链接 SQLite 的客户端

SQLite 网站上有二进制的静态 CLP，但是如果您需要使用 SQLite DLL 版本的 CLP，

可以按照如下步骤构建一个这样的程序：

注意：大部分构建步骤与构建 DLL 类似，前面列出的图片可作为这里的参考。

1. 在主菜单中选择**文件→新建→项目**，在项目类型中选择 Visual C++项目，并选择 Win32。选择 Win32 项目模板，并给项目命名（例如 shell），然后单击 OK。
2. Win32 应用程序向导自动打开。选择应用程序设置，设置应用程序类型为 Console 应用程序，确保选中空项目框。单击完成创建空白的可执行项目。
3. 添加 SQLite CLP 源代码文件。选择**项目→添加**已存在项，在出现的对话框中添加源文件 shell.c。
4. 设置 Visual C++链接到 SQLite DLL。选择**项目→属性**，在出现的对话框的配置下拉框中选择的所有配置，再选择连接器文件夹，选择输入子菜单并停留在其他依赖属性页面，添加 sqlite3.lib。现在可以构建程序了。注意，SQLite DLL 需要和命令行程序在同一目录中，或者在 Windows 系统路径中。

注释：如果构建启用线程的 SQLite DLL 或者从 SQLite 网站获取 DLL，构建 CLP 时需要使用多线程微软 C 运行库 DLL。要这样做，可参考“用 Visual C++构建 SQLite DLL”中的第 4 步下半部分，参考那里的两幅图很容易设置该选项。

用 MinGW 构建 SQLite

MinGW（www.mingw.org）是 Cygwin 项目的分支，它为 Windows 平台提供了一个非常好用的 GUN 编译器集合（GCC）发布版。包含了 Windows 平台指定的头文件和库文件，可以创建本地 Windows 程序，而不用依赖第三方 C 运行 DLL。简单地说，它是 Windows 中免费的 C/C++编译器，并且非常好用。它通常与 MSYS 联合使用，MSYS 提供一个类似 bash 的 Shell，以及让 UNIX 用户在 Windows 中感觉如同在 UNIX 上一样的 POSIX 环境。这两者一起提供了强大的在 Windows 中编译和构建软件的环境。这两者非常流行，同时也很难手动集成。非常感谢有几个非常好的集成完成了这些艰苦的部署工作，使你可以轻易从源代码编译自己的 SQLite。在此推荐使用 TakeOffGW 包，可以从 SourceForge 上下载。

用 MinGW 的 TakeOffGW 发行版和 MSYS 从源代码构建 SQLite DLL，步骤如下：

1. 打开浏览器，进入 SourceForge 网站：www.sourceforge.net。
2. 搜索 TakeOffGW 项目，或者直接进入 sourceforge.net/projects/takeoffgw/。
3. 下载最新版本的 TakeOffGW 网络安装器，通常叫做 setup.exe。
4. 下载 TakeOffGW 网络安装器，运行 setup.exe 并选择默认的安装位置，安装包目录，然后连接到网上（可能需要使用代理服务器）。当出现如图 2-7 所示的包选择页面时，确保选择 MSYS 标题下所有的包。执行操作：选择 MSYS 标题，直到“install”出现。 29

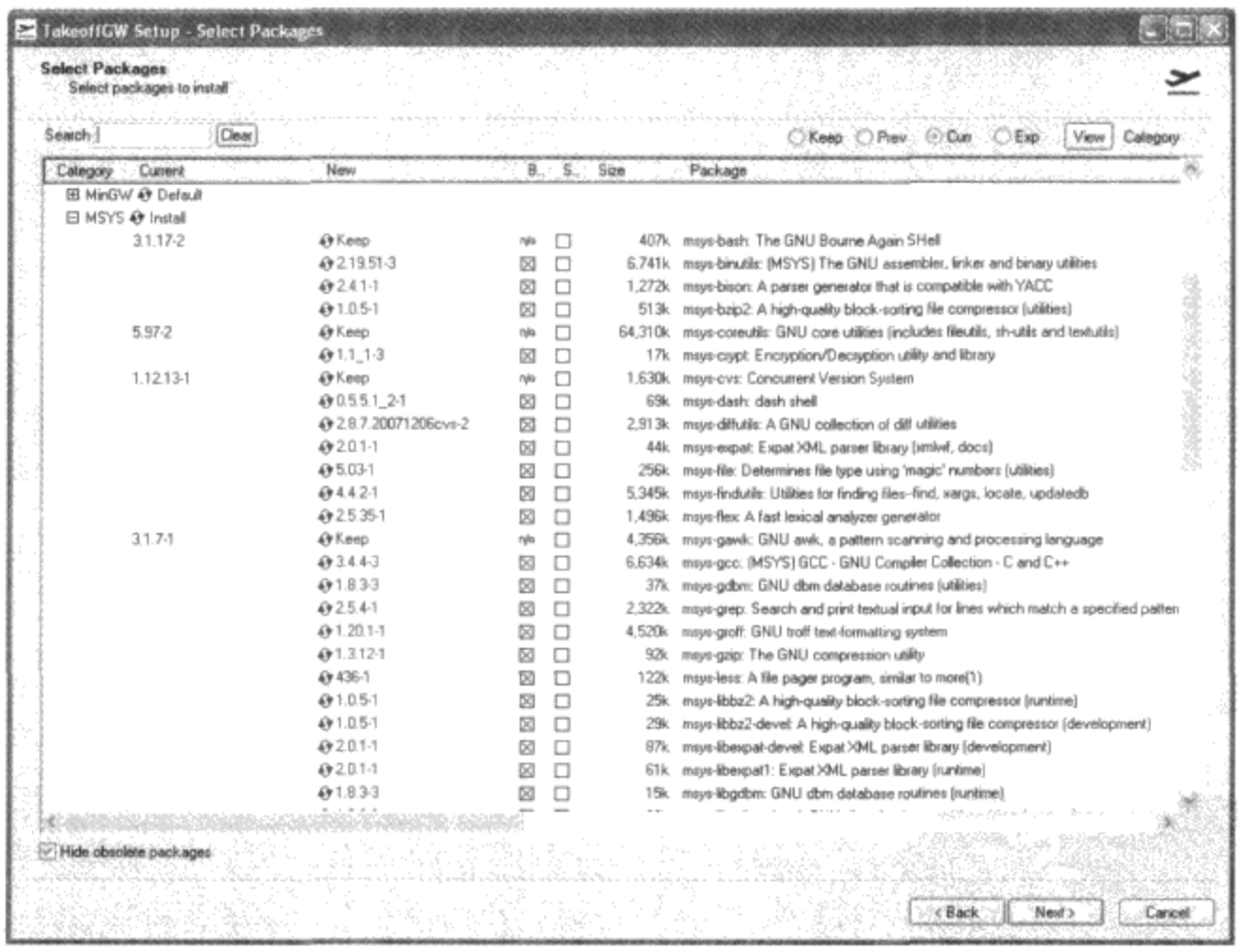


图 2-7　TakeOffGW 安装组件，安装选择 MSYS

5. 单击“下一步”按钮，TakeOffGW 将会下载安装需要的组件。
6. 下载 Linux/UNIX 版本的 SQLite 源代码。打开 www.sqlite.org 网站，单击下载链接，在下载页面中寻找源代码一节。要下载的源代码版本文件名格式为 sqlite-amalgamation-3.x.y.z.tar.Gz，其中，x、y 和 z 代表小版本号（可能还有更小的版本号，本书写作时的版本号是 sqlite-amalgamation-3.6.23.1.tar.gz）。下载 tarball 30

文件并放到临时目录中（例如，C:\Temp）。

7. TakeOffGW 会在桌面生成一个图标，双击该图标打开环境。
8. 进入下载 SQLite 源代码的临时目录。因为 MinGW 是类 UNIX 环境，需要按照 UNIX 文件系统方式使用。例如，进入 C:\temp，应该输入 cd /c/temp。
9. 使用命令，解压 SQLitet tar 文件：tar -xzvf sqlite-amalgamation-3.6.23.1.tar.gz。
10. 进入解压目录：

```
cd sqlite-3.6.32.1
```

11. 创建 Makefile 文件，生成二进制 SQLite，运行如下命令：

```
./configure
```

12. 构建源代码：

```
make
```

你会看到 configure 和 make 会在屏幕上输出很多信息。当 make 信息显示完成后，就会产生一个可以工作的 SQLite 二进制文件。使用 TakeOffGW 非常容易，只需要将 TakeOffGW 的 bin 目录加入到操作系统的 PATH 路径中即可。如果 TakeOffGW 是默认安装的，它的 bin 路径是 c:\cygwin\bin。可以使用资源管理器或者命令行提示符，进入临时目录运行 sqlite3.exe。现在已经有了可以工作的 SQLite CLP。

Linux、苹果 Mac OS X 以及其他 POSIX 系统上的 SQLite

SQLite 在诸如 Linux、Mac OS X、FreeBSD、NetBSD、OpenBSD、Solaris 以及其他 POSIX 系统（虽然 POSIX 现在不是很有用的专业术语，因为很多巨型机和微软 Windows 曾经和 POSIX 兼容十余年，当时 POSIX 非常流行）上的编译和构建基本相同。根据操作系统的不同，可以获得对应的 SQLite 二进制版本。

二进制和包

如果您正在使用苹果 Mac OS 10.4（“Tiger”）或更高版本，系统上就已经安装了 SQLite。如果系统没有默认安装，有多种方式可安装 SQLite。最简单的方式是使用如下的苹果包管理系统，该系统包含了 SQLite 包：

MacPorts：MacPorts 可能是目前 Mac 用户使用的最流行的开源软件、工具和包管理功能的系统。它包含了最新版的 SQLite 和额外的一些包，例如 Tcl 绑定等。

Fink：Fink 是基于 Debian 的包管理系统，它使用 Debian 工具，例如 dpkg dselect 和 apt-get，此外，它也有自己的工具——Fink，读者可以从 http://fink.sourcefourge.Net 下载 Fink。使用 Fink 可以直接从预编译的二进制文件安装，不需要编译步骤。

BSD 用户安装 SQLite 比较容易。FreeBSD、OpenBSD 和 NetBSD 都有 SQLite 包或移植的 SQLite 可用，这些版本都易于安装。在写作本书时，所有的发布版本都有最新的 SQLite 3.6.x 的移植版本。

Solaris 10 将 SQLite 作为操作系统的一部分，最初使用了 SQLite 2.X 版本，但是如果 Solaris 定期更新的话，现在 Solaris 10 使用的是 3.6.20 版本。

正如前面提到的，Linux 平台的二进制文件可以直接从 SQLite 网站下载。SQLite 下载页面提供以下几种二进制格式文件：

静态链接的命令行程序：文件名形式如 sqlite3-3.x.y.z.bin.gz，其中，x、y 和 z 代表小版本号。

共享库：有两种形式的共享库。一种包含 Tcl 绑定；另一种没有包含 Tcl 绑定。有 Tcl 绑定的共享库名称格式为 tclsqlite-3.3.x.y.so.Gz，其中 x 和 y 代表小版本号。两种形式的共享库注释内容是：注意提供的共享库不是线程安全的。如果需要线程安全的版本，则要自己从源代码编译，参考“从源代码编译 SQLite”一节。

SQLite 分析器：SQLite 分析器是提供关于 SQLite 数据库内容的详细信息的命令行程序。读者可以在“获取数据库文件信息”一节查看该程序介绍。

Linux 平台获取最新版本的 SQLite 或者 SQLite 源代码最好的途径是各种相关的包仓库。RedHat、CentOS、Fedora 以及其他 RedHat 衍生版本可以使用 yum 搜索寻找各种 SQLite 包。基于 Debian 的发行版本（包括 Ubuntu）可以轻松获取最新版的 SQLite。使用 apt 或者 Synaptic 在 Ubuntu 和 Debian 软件仓库中搜索 SQLite 3，可在线获取并安装。

从源代码编译

在 Linux、Mac OS X、BSD 以及其他 UNIX 系统中从源代码编译 SQLite 的步骤与前面提到的在 Windows 系统中从 MinGW/TakeOffGW 下编译源代码非常相似（实际上，Windows 系统中的 MinGW 需要更多的步骤，MinGW 是在模拟 Linux 源代码安装）。若

要在这些系统中构建 SQLite，需要确保已经安装了 GNU 编译器集合（GCC），包括 Autoconf、Automake 和 Libtool。大部分系统默认已经安装了这些软件，如果这些软件已经就位，可以按照如下步骤编译 SQLite：

1. 从 SQLite 网站下载 Linux/UNIX SQLite tarball 文件（源代码）。本书写作时的版本为 sqlite-amalgamation-3.6.23.1.tar.gz，保存到某文件夹（例如/tmp 文件夹）。
2. 进入编译文件夹：

```
cd /tmp
```

3. 解压 SQLite tarball 文件：

```
tar -xzvf sqlite-amalgamation-3.6.23.1.tar.gz
```

4. 进入解压文件夹：

```
cd sqlite-3.6.23.1
```

5. 创建 makefile：

```
./configure
```

6. 诸如安装目录之类的其他选项也可以使用。运行如下命令可以获得全部的配置选项。

```
./configure --help
```

7. 编译源代码：

```
make
```

8. 以 root 用户身份安装：

```
make install
```

现在系统中已经安装了可以工作的 SQLite。如果系统安装了 GUN readline，CLP 应该编译成支持 Readline。从命令行运行并测试 SQLite：

```
root@linux # sqlite3
```

这将触发使用内存数据库的 CLP。输入.help ，将输出 shell 命令列表。输入.exit 或按下 Ctrl+D 组合键关闭程序。

命令行程序

SQLite CLP 是使用和管理 SQLite 数据库最常用的工具。它在所有平台上的操作方式相同，因此，学会如何使用 CLP 可以保证永远有一个通用和熟悉的途径来管理数据库。CLP 其实是两个程序，它可以运行在 Shell 模式下以交互的方式执行查询操作，也可以运行在命令行模式下完成各种数据库管理任务。

Shell 模式下的 CLP

打开 Shell，进入临时目录——Windows 系统进入 C:\Temp 或 UNIX 系统进入/tmp。如果在一个固定的目录下工作，可以创建一个 sqlite 目录——Windows 系统是 C:\sqlite，Linux 和 UNIX 系统是/sqlite。这也是当前的工作目录。在与 Shell 一起工作的过程中创建的所有文件都将位于该目录。

■注意：如果想回忆如何获得 Windows 命令行提示符，可参考本章前面的“获得命令行程序”一节第 5 步。

要触发 CLP 的 Shell 模式，可在命令行输入 sqlite3，命令后跟随一个可选的数据库文件名。如果在命令行中不指定数据库名，SQLite 将会使用一个内存数据库（其内容在退出 CLP 时将会丢失）。

CLP 以交互形式运行，可以执行查询、获得 schema 信息、导入/导出数据和其他数据库任务。CLP 会将输入的任何语句当成查询命令，除非命令是以点（.）开始的，这些以点号开始的命令是为指定的 CLP 操作预留的，键入.help 可以得到这些操作的完整列表：

```
fuzzy@linux:/tmp$ sqlite3
SQLite version 3.6.22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .Help
```

```
.backup ?DB? FILE          Backup DB (default "main") to FILE
.bail ON|OFF               Stop after hitting an error. Default OFF
.databases                 List names and files of attached databases
```

```
.dump ?TABLE? ...          Dump the database in an SQL text format
                             If TABLE specified, only dump tables matching
                             LIKE pattern TABLE.
.echo ON|OFF               Turn command echo on or off
.exit                      Exit this program
.explain ?ON|OFF?          Turn output mode suitable for EXPLAIN on or off.
                             With no args, it turns EXPLAIN on.
.genfkey ?OPTIONS?         Options are:
                             --no-drop: Do not drop old fkey triggers.
                             --ignore-errors:Ignore tables with fkey errors
                             --exec: Execute generated SQL immediately
                           See file tool/genfkey.README in the source
                           distribution for further information.
.header(s) ON|OFF          Turn display of headers on or off
.help                      Show this message
.import FILE TABLE         Import data from FILE into TABLE
.indices ?TABLE?           Show names of all indices
                             If TABLE specified, only show indices for tables
                             matching LIKE pattern TABLE.
.load FILE ?ENTRY?         Load an extension library
.mode MODE ?TABLE?         Set output mode where MODE is one of:
                             csv      Comma-separated values
                             column   Left-aligned columns. (See .width)
                             html     HTML <table> code
                             insert   SQL insert statements for TABLE
                             line     One value per line
                             list     Values delimited by .separator string
                             tabs     Tab-separated values
                             tcl      TCL list elements
.nullvalue STRING          Print STRING in place of NULL values
.output FILENAME           Send output to FILENAME
.output stdout             Send output to the screen
.prompt MAIN CONTINUE      Replace the standard prompts
.quit                      Exit this program
.read FILENAME             Execute SQL in FILENAME
.restore ?DB?              FILE Restore content of DB (default "main") from FILE
.schema ?TABLE?            Show the CREATE statements
                             If TABLE specified, only show tables matching
                             LIKE pattern TABLE.
.separator STRING          Change separator used by output mode and .import
.show                      Show the current values for various settings
```

```
.tables ?TABLE?          List names of tables
                           If TABLE specified, only list tables matching
                           LIKE pattern TABLE.
.timeout MS              Try opening locked tables for MS milliseconds
.width NUM1 NUM2 ...     Set column widths for "column" mode
.timer ON|OFF            Turn the CPU timer measurement on or off
.width NUM NUM ...       Set column widths for "column" mode
```

```
sqlite>.exit
```

也可以只输入.h，许多命令都有类似的缩写，例如.e 是退出 shell 命令.exit 的缩写。

命令行模式的 CLP

你可以从命令行使用 CLP，执行如导入/导出数据、返回结果集和一般的批处理工作。使用 Shell 脚本来自动管理数据库是理想的使用方式。从 Shell（Windows 或者 UNIX）上启动 CLP -help，可以看到 CLP 在命令行模式下提供了什么功能：

```
fuzzy@linux:/tmp$ sqlite3 -help
```



```
Usage: sqlite3 [OPTIONS] FILENAME [SQL]
FILENAME is the name of an SQLite database. A new database is created
if the file does not previously exist.
OPTIONS include:
   -help               show this message
   -init filename      read/process named file
   -echo               print commands before execution
   -[no]header         turn headers on or off
   -bail               stop after hitting an error
   -interactive        force interactive I/O
   -batch              force batch I/O
   -column             set output mode to 'column'
   -csv                set output mode to 'csv'
   -html               set output mode to HTML
   -line               set output mode to 'line'
   -list               set output mode to 'list'
   -separator 'x'      set output field separator (|)
   -nullvalue 'text'   set text string for NULL values
   -version            show SQLite version  -init filename    read/process named file
```

命令行中的 CLP 可以接受以下参数：

- 可选列表（可选的）。
- 数据库文件名（可选的）。
- 要执行的 SQL 命令（可选的）。

除了 init 是指定 SQL 命令的批处理文件外，绝大部分选项控制输出格式化。数据库文件名是必需的。SQL 命令是可选的。

数据库管理

现在您已经知道如何触发交互式的 CLP 和命令行模式的 CLP，接下来介绍一个 CLP 执行常见的管理任务的例子。下面从创建数据库开始。

创建数据库

从创建名为 test.db 的数据库开始。启动命令行，通过输入如下命令打开 Shell 模式的 CLP：

```
sqlite3 test.db
```

虽然我们提供了数据库名称，但如果该数据库不存在，SQLite 实际上就未创建该数据库，直到在数据库内部创建一些内容（例如表或者视图）时，SQLite 才创建该数据库。这样做的原因是，让你有机会在数据库结构提交到磁盘前进行各种永久数据库设置（例如页面大小）。数据库一旦创建，一些设置例如页面大小、字符集（UTF-8、UTF-16 等）是不能轻易改变的。因此，在创建数据库前，有机会指定这些参数。这里我们采用默认的设置，要在磁盘中创建该数据库，我们只需要创建一个表，从 Shell 中启动如下语句：


```
sqlite> create table test (id integer primary key, value text);
```

现在磁盘中已经有一个名为 test.db 的数据库文件，数据库包含一个名为 test 的表。正如您看到的，该表有两列：

- 名为 id 的主键列，该列默认具备自动增长的属性。当定义一个整型主键列后，SQLite 会在该列上应用单增函数以便创建一个单增值。也就是说，如果在 INSERT 语句中不提供该列，SQLite 通过查找该列下一个值后自动产生一个。
- 名为 value 的简单文本域。

向表中插入几行数据：

```
sqlite> insert into test (id, value) values(1, 'eenie');
sqlite> insert into test (id, value) values(2, 'meenie');
sqlite> insert into test (value) values('miny');
sqlite> insert into test (value) values('mo');
```

返回插入内容：

```
sqlite> .mode column
sqlite> .headers on
sqlite> select * from test;
```

```
id          value
----------  ----------
1           eenie
2           meenie
3           miny
4           mo
```

select 语句之前的两个命令（.headers 和.mode）用于改善显示格式（这些命令以及类似的命令以后会讲到）。我们可以看到前两行显示提供的 id 值已经使用了，还可以看到 SQLite 为第 3、4 行提供了我们未在 insert 语句中提供的 id 列的顺序整数值。关于自动增量列的主题，您可能会关心最后插入的一条记录的该字段取值，此时可以使用 SQL 函数 last_insert_rowid()获得最后插入的自动增量值：

37

```
sqlite> select last_insert_rowid();
```

```
last_insert_rowid()
-------------------
4
```

退出前，添加一个索引和视图。索引和视图最终都会在后面的例子中用到：

```
sqlite> create index test_idx on test (value);
sqlite> create view schema as select * from sqlite_master;
```

使用.exit 命令退出 Shell。

```
sqlite> .exit
```

在 Windows 系统中也可以使用 Ctrl+C 组合键终止 Shell。在 UNIX 系统中可使用 Ctrl+D 组合键终止。

获得数据库的 Schema 信息

有几个 Shell 命令用于获得有关数据库内容的信息。命令.tables [pattern]可以得到所有表和视图的列表，其中[pattern]可以是任何 like 操作符理解的 SQL（如果你不熟悉 SQL，第 3 章将会介绍）。执行上述命令会返回符合条件的所有表和视图，如果没有提供 pattern 项，则返回所有的表和视图。

```
sqlite> .tables
```

```
schema test
```

可以看到我们创建的表 test 和视图 schema。同样，若要显示一个表的索引，可以键入命令.indices [table name]：

```
sqlite> .indices test
```

```
test_idx
```

可以看到为表 test 所创建的名为 test_idx 的索引。使用.schema [table name]可以得到一个表或视图的定义（DDL）语句。如果没有提供表名，则返回所有数据库对象（包括 table、index、view 和 triger）的定义语句：

```
sqlite> .schema test
```

```
CREATE TABLE test (id integer primary key, value text);
CREATE INDEX test_idx on test (value);
```

```
sqlite> .schema
```

```
CREATE TABLE test (id integer primary key, value text);
CREATE VIEW schema as select * from sqlite_master;
CREATE INDEX test_idx on test (value);
```

更详细的 schema 信息可以通过 SQLite 的重要系统视图 sqlite_master 得到。这个视图是一个系统目录，其结构如表 2-1 所示。

表 2-1　sqlite_master 表结构

编　号	字 段 名	说　明
1	type	对象类型（table、index、trigger、view）
2	name	对象名称
3	tbl_name	对象关联的表

续表

编 号	字 段 名	说 明
4	Rootpage	对象根页面在数据库的索引（开始的编号）
5	sql	对象的 SQL 定义（DDL）

查询当前数据库的 sqlite_master 表，返回如下内容（不要忘记先使用命令.mode column 和.headers on 设置字段格式和标题）：

```
sqlite> .mode col
sqlite> .headers on
sqlite> select type, name, tbl_name, sql from sqlite_master order by type;
type'  name       tbl_name   sql
-------------------------------------------------------
index  test_idx   test       CREATE INDEX test_idx on test (value)
table  test       test       CREATE TABLE test (id integer primary
view   schema     schema     CREATE VIEW schema as select * from s
```


我们看到的 test.db 对象的完整清单包括：一个表、一个索引和一个视图，每一个都有各自最初的 DDL 创建语句。

还有其他几个通过 SQLite 编译指示命令 table_info、index_info、index_list 获取模式信息的命令，第 4 章将会介绍。

提示：别忘了大部分命令行工具（像 SQLite CLP）都会保持您所执行命令的历史记录。要重新运行前面的命令，可以使用向上箭头键浏览以前的命令。在 Windows 系统中，也可以在任何命令提示窗口按 F7 键查看输入命令的列表。

导出数据

使用.dump 命令可以将数据库对象导出成 SQL 格式。不带任何参数时，.dump 将整个数据库导出为数据库定义语言（DDL）和数据操作语言（DML）命令（这些命令会被写入文本或在标准输出上显示），适合重新创建数据库对象和其中的数据。如果您提供了参数，Shell 将参数解析作为表名或视图，导出任何匹配给定参数的表或视图，那些不匹配的将被忽略。在 Shell 模式中，默认情况下，.dump 命令的输出定向到屏幕。如果您要将输出重定向到文件，请使用.dump [filename]命令，此命令将所有的输出重定向到指定的文件中。若要恢复输出到屏幕，只需要执行.output stdout。因此，要将当前数

据库导出到文件 file.sql，可以输入如下命令：

```
sqlite> .output file.sql
sqlite> .dump
sqlite> .output stdout
```

如果文件 file.sql 不存在，将在当前工作目录中创建该文件。如果该文件存在，它将被覆盖。

通过 SQL 的重定向和各种 Shell 格式设置选项（后面会介绍），可以在很大程度上控制导出数据。可以使用您选择的分隔符以及各种格式导出表和视图的具体子集，稍后可以使用下面所述的.import 命令导入。

40 导入数据

有两种方法可以导入数据，用哪种方法取决于要导入的文件格式。如果文件由 SQL 语句构成，可以使用.read 命令导入（执行）文件中包含的命令。如果文件包含由逗号或其他分隔符分隔的值（comma-separated values，CSV）组成，可使用.import [file][table] 命令，此命令将解析指定的文件并尝试将数据插入到指定的表中。它通过使用管道字符（|）作为分隔符解析文件中的每一行，并将已分析的列插入到表中。当然，文件中解析字段的数量应与表中的列数匹配。您可以使用.separator 命令指定不同的分隔符。若要查看分隔符的当前值，可以使用.show 命令。这将显示用户 Shell 中定义的所有设置，当前默认分隔符就在其中：

```
sqlite> .show
        echo: off
     explain: off
     headers: on
        mode: column
nullvalue: ""
      output: stdout
separator: "|"
       width:
```

.read 命令用来导入.dump 命令创建的文件。如果使用前面作为备份文件所导出的 file.sql，需要先移除已经存在的数据库对象（test 表和 schema 视图），然后用下面的方法重新导入：

```
sqlite> drop table test;
sqlite> drop view schema;
sqlite> .read file.sql
```

格式化

CLP 提供了几个格式化选项命令，这些命令可以使结果集和输出更简洁整齐。最简单的是.echo，它将回显输入的命令。.headers 设置为 on 时，查询结果显示时带有字段名。当遇到 NULL 值时，使用.nullvalue 命令设置。例如，如果需要以一个字符串 NULL 来显示 null 值，只需简单执行**.nullvalue NULL** 命令。默认情况下，这种 null 显示时是空串。

如果要改变 CLP 的 Shell 提示符，使用.prompt [value]，如：

```
sqlite> .prompt 'sqlite3> '
sqlite3>
```

.mode 命令可以设置结果数据的几种输出格式。可选的格式有 csv、column、html、insert、line、list、tabs 和 tcl，每种格式都有不同的用途。默认值是 list，list 模式下显示结果集时列间以默认的分隔符分隔。如果想以 CSV 格式输出一个表的数据，可进行如下操作：

```
sqlite3> .output file.csv
sqlite3> .separator ,
sqlite3> select * from test;
sqlite3> .output stdout
```


文件 file.csv 的内容如下所示：

```
1,eenie
2,meenie
3,miny
4,mo
```

实际上，因为 Shell 中已经定义了一个 CSV 模式，所以下面的命令会得到相似的结果：

```
sqlite3> .output file.csv
sqlite3> .mode csv
sqlite3> select * from test;
sqlite3> .output stdout
```

得到的结果是相同的，区别在于 CSV 模式将自动换行字段值并加上双引号，而列表模式（默认）不加。

导出带分隔符的数据

结合导出、导入和设置数据格式的三个部分，有一种以分隔符导出和导入数据的简便方法。例如，要导出 test 表中以字母 m 开始的值并以逗号分隔，导出到 test.csv 文件，可以执行如下操作：

```
sqlite> .output text.csv
sqlite> .separator ,
sqlite> select * from test where value like 'm%';
sqlite> .output stdout
```

如果您要将 CSV 数据导入到与 test 表结构类似的表（称为 test2），可以执行下列操作：

```
sqlite> create table test2(id integer primary key, value text);
sqlite> .import text.csv test2
```

因此，CLP 可以将数据从数据库导出为文本分隔的数据，也可以将文本中分隔的数据轻松地导入到数据库。

执行无人值守维护

到目前为止，CLP 以交互方式执行任务如创建数据库并导出数据。但是，您不想总是被绑到座位上输入这些命令，一个 CLP 命令执行一次。相反，可以使用命令模式批处理运行 CLP 命令。可以使用操作系统内置的调度程序安排这些批处理，使它们在需要的时候运行。

注意：可以自由地以交互方式从命令行中调用 CLP。任何时候有一堆命令要触发执行时，使用命令行方法很有用。

实际上，有两种方法在命令行模式调用 CLP。第一个是提供 SQL 命令或 SQLite shell 命令，如.dump 和.schema。任何有效的 SQL 或 SQLite Shell 命令都可以。SQLite 将执行指定的命令，将结果打印到标准输出并退出。例如，从命令行转存 test.db 数据库，执行

以下命令：

```
sqlite3 test.db .dump
```

应将输出重定向到文件，否则没有什么作用：

```
sqlite3 test.db .dump > test.sql
```

文件 test.sql 现在包含数据库 test.db 中所有可读的 DDL 和 DML 语句。同样，若要选择 test 表的所有记录，执行如下命令：

```
sqlite3 test.db "select * from test"
```

第二种在命令行模式调用 CLP 的方法是将定向到的一个文件作为输入流。例如，若要从数据库 dump 文件 test.sql 中创建新的数据库 test2.db，执行以下操作：

```
sqlite3 test2.db < test.sql
```

CLP 会读取该文件作为标准输入，然后处理其中所有的 SQL 命令，并应用于 test2.db 数据库文件。

从 test.sql 文件创建数据库的另一种方式是使用 init 选项，并将 test.sql 作为参数：

```
sqlite3 -init test.sql test3.db
```

CLP 会处理 test.sql，创建数据库 test3.db，然后进入 Shell 模式。为什么呢？调用没有包含 SQL 命令或输入流。若要避免这种问题，需要提供的 SQL 命令或 SQLite shell 命令，例如：

```
sqlite3 -init test.sql test3.db .exit
```

.exit 命令提示 CLP 运行在命令行模式下，并做尽可能少的工作。考虑到各方面的情况，重定向也许是从命令行处理文件的最简单的方法。

备份数据库

有两种方式可以完成数据库的备份，具体使用哪种取决于您希望的备份类型。SQL 转储也许是移植性最好的备份。生成转储的标准方式是使用 CLP.dump 命令，如在上一节中所示。从命令行可以这样做，如下所示：

43

```
sqlite3 test.db .dump > test.Sql
```

在 Shell 中，可以将输出重定向到外部文件，执行命令，恢复屏幕输出，如下所示：

```
sqlite> .output file.sql
sqlite> .dump
sqlite> .output stdout
sqlite> .exit
```

同样，容易将 SQL 转储作为 CLP 的输入流实现数据库导入：

```
sqlite3 test.db < test.sql
```

这里假定 test.db 尚未存在。如果该数据库存在，只要 test.sql 内容有别于 test.db，操纵仍可工作。当然也会遇到错误，如果 test.sql 包含已驻留在 test.db 的对象或包含违反主键和外键约束的数据（参考后面章节中 PRAGMA 讨论的如何处理这种行为部分）。

备份二进制数据库只是比复制文件稍多做一点工作。备份之前需要清理数据库，这样可以释放一些已删除对象不再使用的空间。这数据库文件就会变小，因此，二进制的副本也会较小：

```
sqlite3 test.db vacuum
cp test.db test.Backup
```

一般而言，备份二进制文件没有备份 SQL 移植性好。总体而言，SQLite 具有很好的向后兼容性，并且所有平台上给定的数据库格式的二进制是兼容的。然而，对于长期备份，使用 SQL 形式才是长久之计。如果文件大小是一个问题，SQL 格式（原始文本）能产生较好的压缩比。

■注意：无论所选的备份方法多完备，一定要记住，能成功恢复的备份才是好的。如果需要依赖备份，就必须测试恢复过程，否则，恢复可能会失败。

最后，如果已经使用过其他数据库，SQLite 中的“删除”数据库就像二进制的备份一样，是一个简单的文件操作：只需删除您要删除的数据库文件即可。

44 获得数据库文件的信息

获取逻辑数据库的信息（例如，表名、DDL 语句等）的主要方法是使用 sqlite_master 视图，该视图提供有关给定数据库包含的所有对象的详细信息。

如果希望了解物理数据库结构的信息，可以使用一种称为 SQLite Analyzer 的工具，

可以从 SQLite 网站下载该工具的二进制包。SQLite Analyzer 提供有关 SQLite 数据库磁盘结构的详细技术信息，这些信息包括数据库、表和索引分类的单个对象，以及聚合的统计信息。它提供从数据库属性如页面大小、表的总数、索引、文件大小和页面平均密度（利用率）到单个数据库对象的详细说明等信息。下面的报告列出了报表中使用的所有术语定义的详细列表。sqlite_analyzer 的部分输出如下所示：

```
fuzzy@linux:/tmp$ sqlite3_analyzer test.db
```

```
/** Disk-Space Utilization Report For test.db
*** As of 2010-May-07 20:26:23

Page size in bytes ····························1024
Pages in the whole file (measured) ·····3
Pages in the whole file (calculated) ·····3
Pages that store data ·····3        100.0%
Pages on the freelist (per header) ·····0        0.0%
Pages on the freelist (calculated) ·····0        0.0%
Pages of auto-vacuum overhead ·····0        0.0%
Number of tables in the database ·····2
Number of indices ·····1
Number of named indices ·····1
Automatically generated indices ·····0
Size of the file in bytes ·····3072
Bytes of user payload stored ·····26       0.85%

*** Page counts for all tables with their indices ********************

TEST ·····2        66.7%
SQLITE_MASTER ·····1        33.3%

*** All tables and indices ***********************************************

Percentage of total database ·····100.0%
Number of entries ·····11
Bytes of storage consumed ·····3072
Bytes of payload ·····235      7.6%
Average payload per entry ·····21.36
Average unused bytes per entry ·····243.00
Maximum payload per entry ·····72
Entries that use overflow ·····0        0.0%
```

```
Primary pages used ·····························3
Overflow pages used ····························0
Total pages used ·····························3
Unused bytes on primary pages ·····2673     87.0%
Unused bytes on overflow pages ·····0
Unused bytes on all pages ·····2673     87.0%

*** Table TEST and all its indices ************************************

Percentage of total database ·····66.7%
Number of entries ·····8
Bytes of storage consumed ·····2048
Bytes of payload ·····60       2.9%
Average payload per entry ·····7.50
Average unused bytes per entry ·····2       43.00
Maximum payload per entry ·····10
Entries that use overflow ·····0       0.0%
Primary pages used ·····2
Overflow pages used ····0
Total pages used ·····2
Unused bytes on primary pages ·····1944     94.9%
Unused bytes on overflow pages ·····0
Unused bytes on all pages ·····1944     94.9%
```

45

SQLite 网站提供 Linux、Mac OS X 和 Windows 平台上二进制版本的 SQLite Analyzer。您可使用 UNIX 提供的 makefile 从源代码编译 SQLite Analyzer。从构建目录中执行以下命令：

```
make sqlite3_analyzer
```

但是，必须在构建配置中设置 Tcl 支持，因为 SQLite Analyzer 使用 Tcl 扩展执行大多数工作。有关的详细信息，请参阅“从源代码编译 SQLite”。

其他 SQLite 工具

有很多其他开源的或商业的程序可与 SQLite 一起工作，其中具备优秀图形化界面且跨平台的有：

- SQLite Database Browser（http://sqlitebrowser.sourceforge.net）：允许用户管理数

据库、表和索引，以及导入和导出这些对象。用户可以交互式运行 SQL 查询并检查结果，以及检查所执行的 SQL 命令日志。最近进行了重大升级，升级到版本 2。

- SQLiteman（http://www.sqliteman.com）：一个针对处理和管理 SQLite 数据库的跨平台程序。它允许一般的数据库、表、索引和触发器以及其他常见的管理任务。 46
- SQLiteManager（www.sqlabs.net/sqlitemanager.php）：一套旨在使用和管理 SQLite 的商业软件。用户可以管理数据库对象、执行查询和保存 SQL，以及用灵活的报表模板创建报表。它包括补充其 SQL 功能的高级脚本语言。

这些都是跨平台的工具。还有更多的工具可用于任何编程、最终用户或管理环境。读者可从 SQLite wiki（www.sqlite.org/cvstrac/wiki?p=ManagementTools）中找到这些软件包的详细信息。

总结

无论使用什么平台，都很容易安装和构建 SQLite。Windows、Mac OS X 和 Linux 用户可以直接从 SQLite 网站下载二进制文件。其他的操作系统用户也可以获得适合本机的二进制文件或第三方安装包系统。

跨平台使用 SQLite 的常用方法是使用 SQLite 命令行程序（CLP）。该程序运行命令行工具以及交互式 Shell，可以发起查询、执行必要的数据库管理任务如创建表、索引和视图以及导出和导入数据。SQLite 数据库包含在单个操作系统文件中，因此，像备份之类的工作就非常简单——只需要复制该文件。不过，对于长期的备份，最好将数据库转储成 SQL 格式，因为这样是跨 SQLite 版本的。

接下来的几章中，您将使用 CLP 开始 SQL 和 SQLite 数据库方面的探索。从第 3 章“SQLite 中的 SQL”基础开始，逐步进入到第 4 章中的“高级 SQL”。

第 3 章

SQLite 中的 SQL

SQL 是数据库的一个重要议题，SQLite 也不例外。本章介绍 SQLite 中的 SQL，全面覆盖了 SQL 的相关知识。不管您是刚接触 SQL 还是 SQL 高手，即使您以前从未使用过 SQL，理解本章内容也不会有障碍。本章主要讨论 SQL，为使内容和结构更清晰、更易于理解，也会附带讲解一些关系概念，但不会深陷理论之中。所以，如果您想绕开支撑 SQLite 和其他关系数据库的关系理论，也不必着急。

SQL 是与关系数据库通信的唯一（也是普遍的）的方法。它专注于信息处理，是为构建、读取、写、排序、过滤、映射、计算、产生、分组、聚集和通常的管理信息而设计的。

SQL 是一种直观的、用户友好的语言，使用有趣并且功能十分强大。比如，不管您是专家还是菜鸟，SQL 总能让你学习到处理事情的新方式（当然有可能更好，也有可能更坏）。

本章的目的是学习如何使用 SQL——介绍一些好技术和便捷的技巧。从目录中您已经知道 SQLite 中的 SQL 是一个非常大的主题，我们将其分为两章，第 3 章覆盖了核心的 SELECT 语句，第 4 章讲解其他部分。但是介绍完本章内容后，您应该可以操作任何数据库了。

注意：第 3 章的内容只是集中在 SQLite 中 SQL 的使用上。要想深入学习 SQL，请参考 Claire Churcher 的《从 SQL 查询开始》（*Beginning SQL Queries*）。

数据库示例

在开始讲解语法前，我们先熟悉本书中的一些示例数据库。本章使用的数据库（本

书其他章节也使用该数据库）由辛菲尔德（Seinfeld）的各系列食物组成，您可能曾留意过 Seinfeld。根据网上数据，辛菲尔德历史上有 180 个系列、400 多种不同的食品。每次展示都会展出两种以上的新食品，减去商业时间，几乎是每 10 分钟引入一种新食品。

48 因此，这种对食物的专注当然值得提倡，何况它还很好地组成一个说明 SQL 必要概念和使用 SQLite 中 SQL 的数据库。图 3-1 显示了示例数据库中的表。

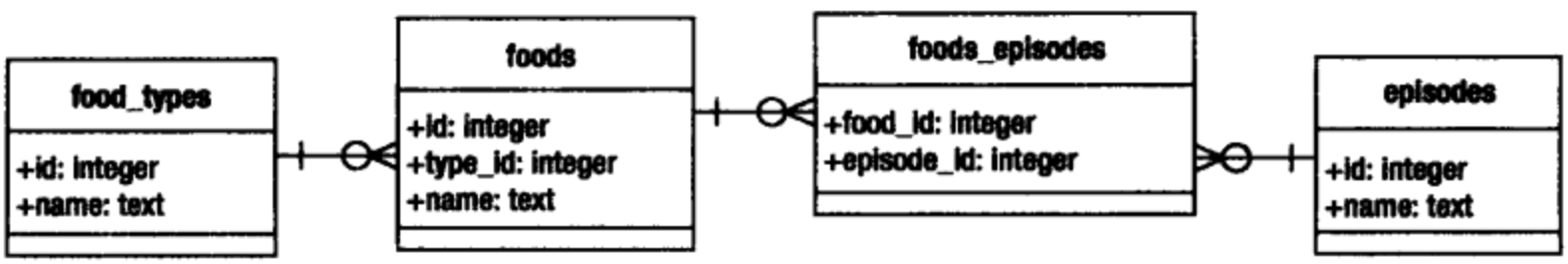


图 3-1　辛菲尔德食物数据库

SQLite 中定义的数据库模式如下：

```
create table episodes (
  id integer primary key,
  season int,
  name text );

create table foods(
  id integer primary key,
  type_id integer,
  name text );

create table food_types(
  id integer primary key,
  name text );

create table foods_episodes(
  food_id integer,
  episode_id integer );
```

主要的表是 foods 表。foods 表中每一行都对应一种不同的食物，名字存储在 name 字段中。type_id 引用 food_types 表，用来存储各种食物的分类（例如，烘烤食物、饮料、垃圾食品）。最后，foods_episodes 表将 foods 表的 foods 和 episodes 表的 episodes 连接起来。

安装

食物数据库位于本书附带例子的 zip 文件中，您可以从 Apress 网站（www.apress.com）的源代码部分下载。为了创建该数据库，首先进入解压缩文件的根目录寻找 foods.sql，从命令行开始创建新数据库，进入示例目录，执行如下命令：

```
sqlite3 foods.db < foods.sql
```

将会创建一个名为 foods.db 的数据库。如果您不熟悉如何使用 SQLite 命令行程序，可参考第 2 章。 49

运行示例

为方便起见，本章所有的例子都可以在 examples.zip 文件的根目录下面的 sql.sql 文件中找到。不用手动输入，只需要打开文件定位到需要的 SQL 就可以。

本章中运行较长查询的简单方法是将这些查询复制到您常用的编辑器中，保存成单独的可以从命令行运行的文件。例如，要从 test.sql 中复制一个很长的查询，可以使用与前面创建数据库同样的方法运行以下命令：

```
sqlite3 foods.db < test.Sql
```

结果会输出在屏幕上。这也使得试验这些查询更容易，不用从 SQLite shell 中重新输入或者编辑它们。只需要在编辑器中修改并保存文件，然后在命令行中重新运行即可。

为了使数据内容更具可读性，可以在文件的开始加上以下命令：

```
.echo on
.mode column
.headers on
.nullvalue NULL
```

这些命令使得命令行程序：1）在屏幕上打印执行的 SQL 语句，2）以列模式显示结果，3）包含列名称，4）将 nulls 打印成 NULL。本章中所有例子的数据都用具体的设置进行格式指定。另一个用来设置的选项是.width 选项，该选项设置输出结果中各个列的宽度，这些设置因例子各异。

为了提高输出的可读性，下面以两种不同的格式展示这些例子。对于较短的查询，我们将以 SQLite Shell 中显示的方式，显示 SQL 和输出结果。例如：

```
sqlite> select *
   ...> from foods
   ...> where name='JujyFruit'
   ...> and type_id=9;
id          type_id     name
----------  ----------  ----------
244         9           JujyFruit
```

如前面的例子，有时我们可以在命令和输出结果之间加些额外内容以改善可读性。对于较长的查询，我们以代码形式显示 SQL 并与结果之间用灰线隔开，如下所示：

```
50  select f.name name, types.name type
    from foods f
    inner join (
      select *
      from food_types
      where id=6) types
    on f.type_id=types.id;
```

```
name                        type
--------------------------  -----
Generic (as a meal)         Dip
Good Dip                    Dip
Guacamole Dip               Dip
Hummus                      Dip
```

语法

SQL 的声明式语法读起来就像自然语言，以命令模式表达语句，以描述行为的动词开始。下面是主语和谓语，如图 3-2 所示。

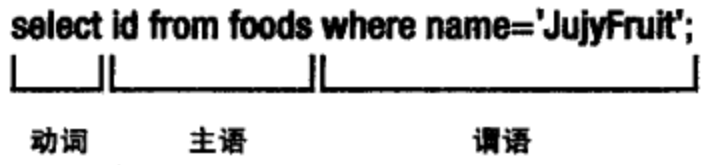


图 3-2　SQL 语法的一般结构

正如您所看到的，就像一个普通语句一样，SQL 是专门为那些非技术人员设计的，因此非常简单且易于理解。

SQL 的易用性部分来源于它是（绝大部分是）声明式语言，而不是命令式语言，例如 C、Perl。声明式语言是您可以描述想要什么，而命令式语言是那种您需要指定如何做的语言。例如，以处理干酪汉堡包订单的过程为例，您作为客户，使用声明式语言说出您的订单，即对前台人员说：

给我一份含双份肉的，外加墨西哥胡椒和奶酪的干酪汉堡包，抹上蛋黄酱。

订单传递给厨师，他使用命令式语言编写的程序填写订单——食谱。根据您的声明式细则，遵照一系列已经定义好的必须执行的步骤，创建干酪汉堡包的具体订单： 51

1. 从左边的第三个冰箱取出碎牛肉。
2. 做小肉饼。
3. 烹 3 分钟。
4. 翻转。
5. 烹 3 分多钟。
6. 重复步骤 1～5 做第二块小肉饼。
7. 给上面的面包涂上芥末。
8. 将肉饼放到下面的面包上。
9. 给汉堡加上奶酪、生菜、番茄酱、洋葱和墨西哥胡椒。
10. 将上下两块面包合起来，用黄纸包上。

正如您看到的，声明式语言比命令式语言更简洁。本例中，采用了声明式汉堡语言（DBL）一步就可以完成干酪汉堡包的具体工作，而命令式厨师语言（ICL）需要 10 步。声明式语言以较少的声明完成较多的工作。事实上，SQL 的易用性不局限于本例。与前面我们想象的 DBL 语句等价的 SQL 表示可能如下：

```
select burger
from kitchen
where patties=2
and toppings='jalopenos'
and condiment != 'mayo'
limit 1;
```

非常简单。正如我们提到的，SQL 开始就是设计成用户友好语言。早期，SQL 是为那些执行例如及时查询和生成报表工作的终端用户准备的（而不是像现在几乎只有开发者和数据库管理员在用 SQL）。

命令

SQL 由命令组成，每条命令以分号（;）结束。例如，下面是 3 条独立的命令：

```
select id, name from foods;
insert into foods values (null, 'Whataburger');
delete from foods where id=413;
```


■注意：SQLite 用分号作为命令终结符，启动处理命令。命令终结符通常与交互式程序相关，是用户用来对数据库执行查询的。分号是许多系统上通用的 SQL 命令终结符，虽然有些数据库使用\g 或者单词 go。

反过来，命令是由一系列记号组成的。记号可以是常量、关键字、标识符、表达式或者特殊字符。令牌符号以空格分开，例如空格、tab 和新一行。

常量

常量也称为 constants，表示确切的值。它包含 3 种类型：字符串常量、数字常量和二进制常量。字符串常量由单引号引起来的一个或多个字母或数字字符组成。例如

```
'Jerry'
'Newman'
'JujyFruit'
```

虽然 SQLite 支持以单引号或双引号界定字符串，但是我们强烈建议您只使用单引号。因为这是 SQL 标准，如果您遇见了严格要求字符分界的系统，这将节省很多时间。如果字符串中本身包含单引号，需要连续两个单引号，例如，Kenny's chicken 需要写成：

```
'Kenny''s chicken'
```

数字常量有整数、十进制数和科学记数法表示的数，下面是一些例子：

```
-1
3.142
6.0221415E23
```

二进制值使用 x'0000'的表示方法，其中每位是一个 16 进制数。二进制值必须由两个 16 进制数的整数倍（8 bits）组成，下面是一些例子：

```
x'01'
X'0fff'
x'0F0EFF'
X'0f0effab'
```

关键字和标识符

关键字是 SQL 中有指定含义的单词，这些单词包括 select、update、insert、create、drop 和 begin 等。标识符是指数据库里的具体对象，如表或索引。关键字是保留的单词，不能用做标识符。SQL 不区分关键字和标识符的大小写，例如，下面是相同的语句：

```
SELECT * from foo;
SeLeCt * FrOm FOO;
```

SQLite 中默认的字符常量值是大小写敏感的。因此，'Mike' 与'mike'是不同的。

注释

SQL 中的单行注释是用两个连续连字符（--）表示的。多行注释使用的是 C 语言风格的/* */形式。例如：

```
-- This is a comment on one line
/* This is a comment spanning
   two lines */
```

再次强调，除非您有十分充分的理由使用 C 语言风格的注释，否则我们推荐在 SQLite 的 SQL 脚本中使用 SQL 标准的两个连续连字符。

创建数据库

表是探索 SQLite 中 SQL 的起点，也是关系数据库中信息的标准单元。所有的操作都以表为中心，表由行和列组成。虽然听起来很简单，但是表与各种其他概念相关，这些无法用三言两语解释清楚。事实上，表几乎占据了本章大部分内容。我们即将要做的就是花两分钟纵览一下表——只需要能创建简单的表并能处理它就行。一旦完成表的创建，本章其他部分将会用到。

创建表

像关系模型一样，SQL 是由几个部分组成的。它有结构化部分，例如，用来创建和销毁数据库对象的部分，这一部分语言通常称为数据库定义语言（DDL）。类似地，还有一些在这些对象上执行操作的功能部分（例如获取和控制数据），这一部分语言称为数据操作语言（DML）。创建表的语句属于 DDL 部分，可以用如下定义的 create table 命令创建表：

```
create [temp] table table_name (column_definitions [, constraints]);
```

54

用 temp 或 temporary 关键字声明的表是临时表，这种表是临时的——只存活于当前会话。一旦连接断开，就会被自动销毁（如果您没有手动销毁）。temp 旁边的方括号表示该项是可选项。任何时候您看见方括号中的语法，它都表示方括号的内容是可选项。此外，管道符号（|）表示两者选其一（相当于单词或者）。例如，如下语法：

```
create [temp|temporary] table … ;
```

这意味着 temp 或者 temporary 关键字是可选的。您可以写成 create temp table foo…，也可以写成 create temporary table foo…。这两种语句实质是相同的。

如果没有明确指出创建临时表，则创建的是基本表，将会在数据库中持久存在。这也是数据库中最常见的表。一般而言，基本表用来区分由 create table 创建的表与系统表，以及其他类似表的对象，例如视图。

create table 命令至少需要一个表名和一个字段名，命令中的 table_name 表示表名，表名必须与其他所有的标识符不同。column_definitions 由用逗号分隔的字段列表组成，每个字段定义包括一个名称、一个域和一个逗号分隔的字段约束。类型有时也称为“域”，在编程语言中，它们是同义词，都是指存储在该列的信息的类型。

在 SQLite 中有 5 种本地类型：integer、real、text、blob 和 null，所有这些域将在本章后面的“存储类”一节中介绍。“约束”用来控制什么样的值可以存储在表中或特定的字段中。例如，您可以用 unique 约束来规定所有的记录中某个字段的值要各不相同。约束将会在“数据完整性”一节中介绍。

Create table 命令允许您在字段列表后面跟随一个附加的字段约束，如下例：

```
create table contacts ( id integer primary key,
                        name text not null collate nocase,
                        phone text not null default 'UNKNOWN',
```

```
        unique (name,phone) );
```

字段 id 声明为 integer 型，限制为主键。这种组合在 SQLite 中有特殊含义，整型主键基本上表示该字段是自增长字段（本章后面的“主键限制”一节会介绍）。字段 name 声明为 text 类型，约束不能为 null，并且排序不区分大小写。字段 phone 是 text 类型，也有两条约束。之后，表一级的约束是 unique，定义在字段 name 和 phone 上。

有很多内容要学习，但是后面的内容会进一步解释。您可能感觉到表有多复杂，其实这里唯一重要的事情就是理解 create table 语句的一般格式。

修改表

您可以用 alter table 命令改变表的部分结构。SQLite 版的 alter table 命令既可以改变表名，也可以增加字段。该命令的一般格式为：

```
alter table table { rename to name | add column column_def }
```

注意，这里又出现了新的符号{}。花括号括起来一个选项列表，表示必须从各选项中选择一个。此处，我们使用 alter table table rename...，或者 alter table table add column...。也就是说，您可以使用 rename 子句重命名表，也可以使用 add column 子句添加列。重命名表可以通过提供新的 name 完成。

55

如果想添加新字段，column_def 表示字段定义遵从 create table 中的形式。即，字段名后跟一个可选的类型和约束列表。例如：

```
sqlite> alter table contacts
        add column email text not null default '' collate nocase;
sqlite> .schema contacts

create table contacts ( id integer primary key,
                        name text not null collate nocase,
                        phone text not null default 'UNKNOWN',
                        email text not null default '' collate nocase,
                        unique (name,phone) );
```

在 SQLite 命令行程序中查看表定义，可使用.schema 命令后面紧跟表名称，当前表的定义就会输出。如果不提供表名，.schema 会打印出整个数据库的模式。

表还可以由 select 语句的结果创建，这样在创建表结构的同时创建数据。这种特别

的 create table 语句将在后面“插入记录”一节中介绍。

数据库查询

设计数据库和创建表的最终目的是使用数据。与数据一起工作是 SQL 数据操作语言的工作（DML）。DML 的核心是 select 命令，它也是查询数据库的唯一命令。select 是 SQL 命令中最大、最复杂的命令。select 的很多操作都来源于关系代数，并且包含了关系代数的很多内容。即使在 SQLite 这样精简而高效的环境中，select 语句的能力和复杂性依然是巨大的。但是不要因此而走开，SQLite 通向 select 命令的方法非常具有逻辑性，它具备支持关系数据库的底层关系理论的坚实基础。

关系操作

从理论上讲，以某种概念的方式考虑一下 select 做什么以及为什么是有用的。大部分 SQL 实现中，包括 SQLite，select 语句提供混合、比较和过滤数据的“关系操作”，这些关系操作通常划分为 3 种类型。

- 基本操作
 - — Restriction（限制）。
 - — Projection（投影）。
 - — Cartesian Product（笛卡儿积）。
 - — Union（联合）。
 - — Difference（差）。
 - — Rename（重命名）。
- 附加操作
 - — Intersection（交叉）。
 - — Natural Join（自然连接）。
 - — Assign（赋值）。
- 扩展操作
 - — Generalized Projection（广义投影）。
 - — Left Outer Join（左外连接）。

— Right Outer Join（右外连接）。

— Full Outer Join（全外连接）。

基本操作定义了基础的关系操作，并在集合论中都有相应的理论基础。附加操作是为了方便，提供了频繁执行操作的快速方式，这些操作是由基本操作组成的。例如，交叉操作可以通过如下操作实现，取两个集合的联合，通过区分操作删除两个结果，进一步联合有彼此差异的两个初始集合。最后，扩展操作是对基本操作和附加操作的扩展。例如，广义投影操作在基本的映射操作上添加算术表达、聚集和分组功能。外连接扩展了连接操作，允许从数据库获取额外的信息和/或不完全的信息。

标准 ANSI SQL 中，select 可以完成上述所有的关系操作。这些操作覆盖了 E.F Codd 在关于关系理论的最初工作中定义的所有关系运算符（只有一个例外——divide）。SQLite 支持 ANSI SQL 中除 right 和 full outer join 之外的所有操作，这些操作可用其他基本的关系操作完成，所以不用担心缺乏右外连接和全外连接。

所有的这些操作都定义在关系中，也就是通常所说的表。它们将一个或多个关系作为输入，然后产生另一个关系作为输出，它允许捆绑进关系表达式的操作。关系表达式可以创建得极其复杂，例如，选择操作（一种关系）的输出可以是另一个 select 语句的输入，如下面所示：

```
select name from (select name, type_id from (select * from foods));
```

可以看到，最里层的 select 的结果作为次里层的 select 的输入，而次里层的 select 的结果又作为最外面的输入，这是单一关系型表达式。任何熟悉 Linux、UNIX 或 Windows 上管道命令的人都能理解这种行为。任何 select 语句的输出都可以作为另一个语句的输入。

select 命令与操作管道

57

从语法上说，select 命令用一系列子句将很多关系操作组合在一起，每个子句代表一种特定的关系操作。SQLite 中，几乎所有这些子句都是可选的，事实上，只有标准的 SQL 规则是必需的。作为 SQLite 中 SQL 的使用者，这使得您可以只选择获取数据所需要的操作。

select 命令的通用形式如下：

```
select [distinct] heading
```

```
from tables
where predicate
group by columns
having predicate
order by columns
limit count,offset;
```

每个关键字如：from、where、having 等都是一个单独的子句，每个子句由关键字和跟随的参数（例子中的斜体字）构成。这里，我们将通过简化名字引用这些子句，例如“where 子句”，我们使用 where 代替。

在 SQLite 中，理解 select 命令的最好方法就是将其当成处理关系的管道，该管道有可选的流程，可以根据需要选择。不管是否包含或排除某个操作，相关的操作总是相同的，图 3-3 显示了这些顺序。

select 命令从 from 开始，接收一个或多个输入关系；将它们组合成单一复合关系；传递给后续操作链。

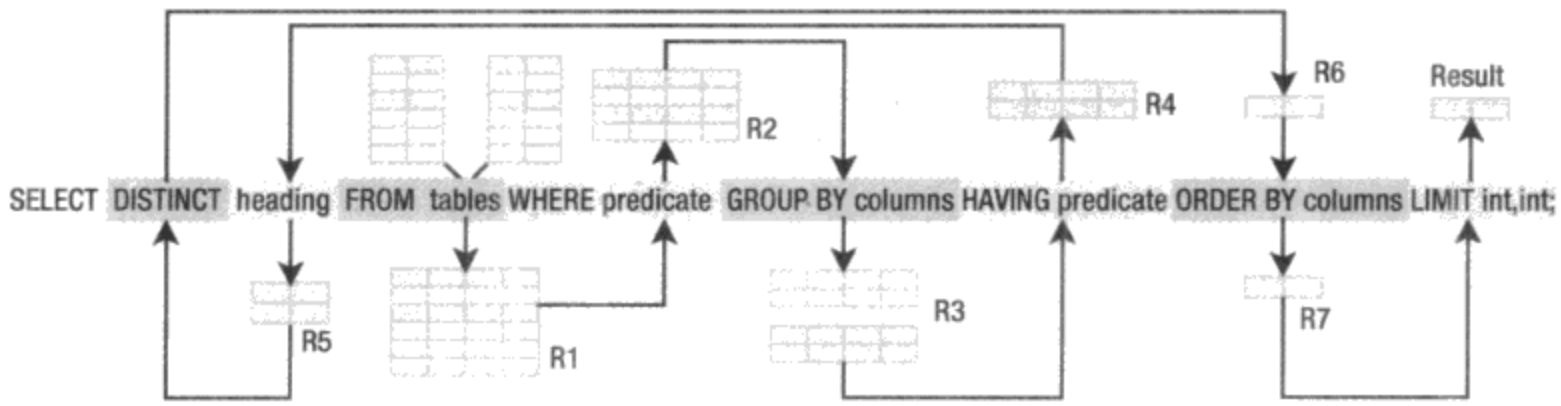


图 3-3　SQLite 中 select 处理过程

除了 select 之外，所有的子句都是可选的，必须提供 select，这样才是有效的 select 命令。目前最常见的 select 命令由三个子句组成：select、from、where。这种基本的语法形式和相关的子句如下所示：

```
select heading from tables where predicate;
```

58 from 子句是由逗号分隔的一个或多个表、视图等（见图 3-3 中变量 tables）。如果指定多个表，它们会组合形成单一关系（见图 3-3 中 R1 所示），这是通过链接操作完成的。from 产生的结果关系作为最初的数据关系，所有后续的其他操作将直接从最初的数据中工作或者从最初数据的派生内容中工作。

where 子句过滤 R1 中的行。where 参数是 predicate，或者是逻辑表达式，它们定义了 R1 中行的选择标准，那些行将会出现（或排除）在结果中。从 where 子句中选中的行形成新的关系 R2，如图 3-3 所示。

在本例中，R2 绕过非重要的操作直到 select 子句。select 在 R2 中过滤列，如图 3-4 所示，它的参数由逗号分隔的列或者定义结果的表达式组成，这也称为投影列表。

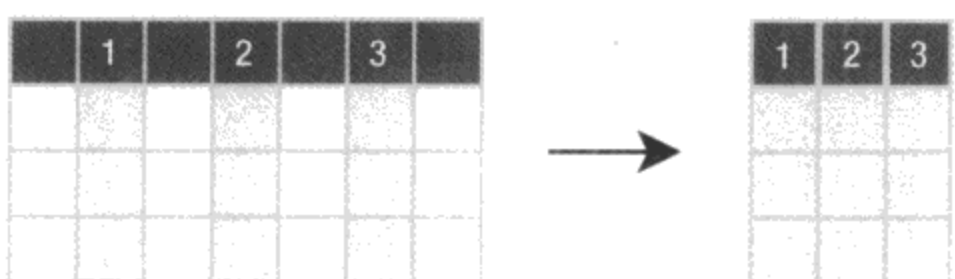


图 3-4　投影

下面是使用本书示例数据库的具体例子：

```
sqlite> select id, name from food_types;
id          name
----------  ----------
1           Bakery
2           Cereal
3           Chicken/Fowl
4           Condiments
5           Dairy
6           Dip
7           Drinks
8           Fruit
9           Junkfood
10          Meat
11          Rice/Pasta
12          Sandwiches
13          Seafood
14          Soup
15          Vegetables
```

没有 where 子句过滤行，所以表 food_types 中所有的行都返回了。select 子句指定表 food_types 的所有列，form 子句没有连接表，结果就是表 food_types 的一份副本。正如绝大部分 SQL 实现一样，SQLite 也支持用星号（*）表示的所有列。因此，前面的例

子也可以简单表示成：

```
select * from food_types;
```

59 总之，SQLite 中基本的 select 处理收集 from 子句中的所有数据，在 where 子句中过滤行，在 select 子句中过滤列（投影）。如图 3-5 所示，显示了其处理流程。

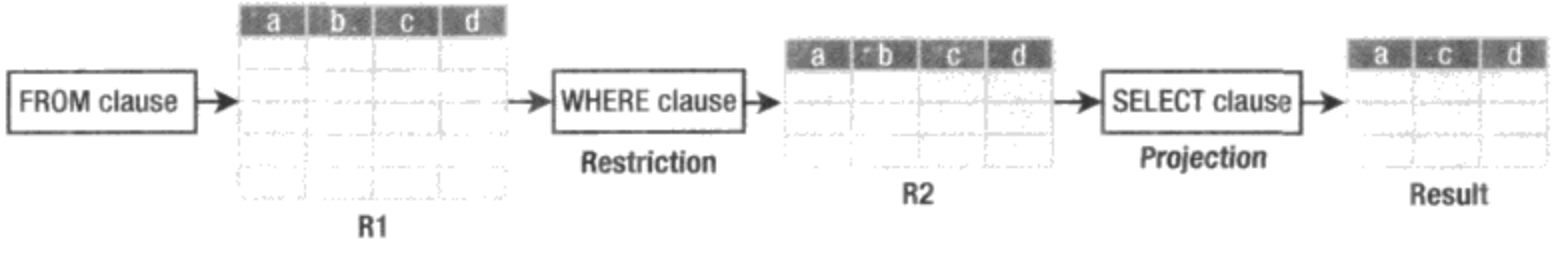


图 3-5　select 中的限制与投影

在这个简单的例子中，您可以看到一般查询语言和特定的 SQL 如何执行关系操作。这些背后都是数学理论。

过滤

如果 select 是 SQL 中最复杂的命令，那么 where 就是 select 中最复杂的子句。但是它也能完成最多的工作，对 where 有了坚实的理解后，将有助于您在 SQLite 中使用 SQL。

SQLite 对 from 子句（R1）产生的每一行应用 where 子句。正如前面谈到的，where 是一种限制，也就是一种过滤器。where 的参数是逻辑预测，最简单的预测形式就是断言某事。考虑如下例子：

```
The dog (subject) is purple and has a toothy grin (predicate).
```

Dog（狗）是主语，预测是由两个断言组成的（颜色是 purple（紫色的），a toothy grin（咧嘴时牙齿很多））。这种声明可能是真，也可能是假，依赖于狗的实际情况。

where 子句的主语是行，行是最大的逻辑主语。where 子句是逻辑预测，评估为真的所有行将会包含在（或选中）结果集（R2）中，评估为假的将被排除。因此，对狗的讨论翻译成等价的关系操作，形式如下：

```
select * from dogs where color='purple' and grin='toothy';
```

数据库会获取表 dogs（主语）中的所有行，然后应用 where 子句（预测）形成逻辑预测：

```
This row has color='purple' and grin='toothy'.
```

如果为真，该行（该狗）的确是紫色的且多牙齿，将会包含在结果集中。where 子句过滤功能十分强大，它能在很大程度上控制条件，通过这些条件可以将一些行包含在结果集或者从结果集中排除。

值

“值”代表了真实世界的某种数据。值可以按它们所属类型划分，如数字值（1、2、3 等）或字符串值（“Jujy-Fruit”）。值可以表现为文字的值（1、2、3 或“JujyFruit”）、变量（一般是如 foods.name 的列名称）、表达式（3+2/5）或函数的结果（count(foods.name)），这些后面会讲到。

操作符

操作符使用一个或多个值作为输入并产生一个新值作为输出。之所以叫“操作符”，是因为它完成某种操作并产生某种结果。二元操作符操作两个输入值（或称操作数），三元操作符操作三个操作数，单目操作符操作一个操作数，如此等等。

操作符可以捆绑在一起，一个操作符的输出可以作为另一操作符的输入（见图 3-6），形成值表达式。

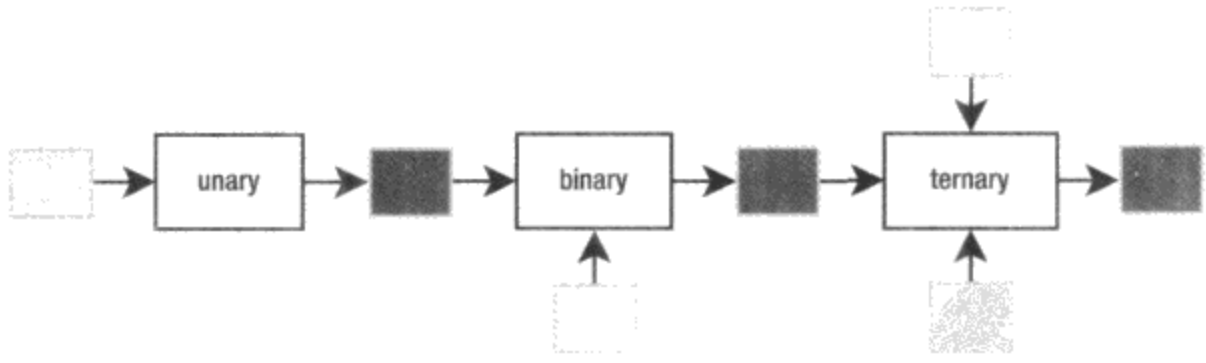


图 3-6　单目、二元和三元操作符以及操作管道

将操作符串接在一起，可以创建值表达式，以其他值表达极其复杂的表达式，例如：

```
x = count(episodes.name)
y = count(foods.name)
z = y/x * 11
```

二元操作符

二元操作符是目前最常用的 SQL 操作符。表 3-1 列出了 SQLite 所支持的二元操作

符，表中按优先级从高到低的次序排列，同色的一组具有相同的优先级。

61

表 3-1　二元操作符

操 作 符	类　　型	作　　用
\|\|	String	连接
*	Arithmetic	乘
/	Arithmetic	除
%	Arithmetic	模
+	Arithmetic	加
-	Arithmetic	减
<<	Bitwise	右移
>>	Bitwise	左移
&	Logical	与
\|	Logical	或
<	Relational	小于
<=	Relational	小于或等于
>	Relational	大于
>=	Relational	大于或等于
=	Relational	等于
==	Relational	等于
<>	Relational	不等于
!=	Relational	不等于
IN	Logical	In
AND	Logical	与
OR	Logical	或
62 IS	Logical Equal	等于
LIKE	Relational St	字符串匹配
GLOB	Relational Fi	文件名匹配

算术操作符（例如，加、减、除）都是输入数字值并产生一个数字值的二元操作符。关系操作符（例如，>、<、=）是比较值和值表达式并产生逻辑结果（也称为真假值）的二元操作符。关系操作符形成逻辑表达式，例如：

```
x > 5
1 < 2
```

逻辑表达式是返回真假值的任意表达式。在 SQLite 中，false 可以由数字 0 代替，

true 可以由其他的非 0 值代替。例如：

```
sqlite> SELECT 1 > 2;
1 > 2
----------
0

sqlite> SELECT 1 < 2;
1 < 2
----------
1

sqlite> SELECT 1 = 2;
1 = 2
----------
0

sqlite> SELECT -1 AND 1;
-1 AND 1
----------
1
```


逻辑操作符

逻辑操作符（AND、OR、NOT、IN）是对真假值或逻辑表达式进行操作的二元操作符。它们根据输入产生具体的真假值。它们可以用较简单的表达式构建更复杂的逻辑表达式，例如：

```
(x > 5) AND (x != 3)
(y < 2) OR (y > 4) AND NOT (y = 0)
(color='purple') AND (grin='toothy')
```

真假值遵从通常的布尔逻辑规则，但是考虑到 null 就会有点纠结，本章后面会讨论到。where 子句是由这些真假值组成的——使用逻辑操作符回答 SQLite 数据库中数据的真实问题。例如：

```
sqlite> select * from foods where name='JujyFruit' and type_id=9;

id          type_id     name
----------  ----------  ----------
244         9           JujyFruit
```

这里的限制是根据表达式（name='JujyFruit'）和（type_id=9）进行的，由两个通过逻辑与组合的逻辑表达式组成。对于任何 foods 中的记录，只有这两个条件都为真，才可以进入结果集中。

LIKE 与 GLOB 操作符

一个很有用的关系操作符就是 LIKE。LIKE 的作用与相等（=）类似，都是通过一个模式来进行字符串匹配的。例如，要查询表 foods 中所有名称以字符“J”开始的食品，可使用如下语句：

```
sqlite> select id, name from foods where name like 'J%';

id          name
----------  --------------------
156         Juice box
236         Juicy Fruit Gum
243         Jello with Bananas
244         JujyFruit
245         Junior Mints
370         Jambalaya
```

模式中的百分号（%）可与任意 0 个或多个字符匹配。下画线（_）可与任意单个字符匹配。百分号是贪婪匹配，它会匹配除了该字符本身外的所有在字符之间的内容。如果百分号在模式的最左边或者最右边，它将匹配字符的另外一边。参看如下例子：

64
```
sqlite> select id, name from foods where name like '%ac%P%';

id          name
----------  --------------------
127         Guacamole Dip
168         Peach Schnapps
198         Mackinaw Peaches
```

另一个有用的窍门是使用 NOT 否定某些模式：

```
sqlite> select id, name from foods
        where name like '%ac%P%' and name not like '%Sch%'

id          name
----------  --------------------
38          Pie (Blackberry) Pie
```

```
127       Guacamole Dip
198       Mackinaw peaches
```

GLOB 操作符在行为上与 LIKE 操作符非常相似。关键的不同在于它有些像 UNIX/Linux 文件名替换语法。也就是说，它会使用文件名替换相关的通配符，例如*和_，并且匹配是大小写敏感的。下面展示了 glob 行为：

```
sqlite> select id, name from foods
   ...> where name glob 'Pine*';

id          name
----------  ----------
205         Pineapple
258         Pineapple
```

SQLite 也识别 match 和 regexp 正则表达式断言，但是目前不提供自身的实现。您需要开发自己的 sqlite_create_funtion()调用，第 5 章会讨论该内容。

限定和排序

可以用 limit 和 offset 关键字限定结果集的大小和范围。limit 指定返回记录的最大数量，offset 指定偏移的记录数。例如，下面的命令是使用 limit 和 offset 返回 food_types 表中 id 排第 2 的 Cereal 记录：

```
select * from food_types order by id limit 1 offset 1;
```

关键字 offset 在结果集中跳过一行（Bakery），关键字 limit 限制最多返回一行（Cereal）。

上面的语句中还有一个 order by 子句，它使记录集在返回之前按一个或多个字段的值进行排序。本例中的 order by 很重要，因为您无法保证从 select 返回的行是按照某种顺序排列的——标准 SQL 声明了这一点。因此，当您需要依赖某种排序返回结果时，order by 子句很重要。order by 子句的语法与 select 子句语法相似：以逗号分隔的一系列字段，每个字段项都可能配合排序方向——asc（默认的升序）或 desc（降序）。例如：

65

```
sqlite> select * from foods where name like 'B%'
        order by type_id desc, name limit 10;

id     type_id    name
```

```
-----  --------  -------------------
382    15        Baked Beans
383    15        Baked Potato w/Sour
384    15        Big Salad
385    15        Broccoli
362    14        Bouillabaisse
328    12        BLT
327    12        Bacon Club (no turke
326    12        Bologna
329    12        Brisket Sandwich
274    10        Bacon
```

当第一个排序出现重复，一般需要按照第二、第三、……字段去排序。例如，本例中 type_ids 有很多重复的，但我们先将它们按组排在一起，然后在组内按照 foods 字母顺序排序。

■注意：limit 和 offset 不是 ANSI 标准中定义的标准 SQL 关键字。虽然使用的语法有所不同，但是绝大多数数据库都有同等功能。

limit 和 offset 一起使用时，可以用逗号替代 offset 关键字。例如：

```
select * from foods where name like 'B%'
order by type_id desc, name limit 1 offset 2;
```

可以有如下同等表达：

```
sqlite> select * from foods where name like 'B%'
        order by type_id desc, name limit 2,1;

id     type_id   name
-----  --------  -----------------
384    15        Big Salad
```

有些人可能认为这里的语法应该倒过来。在 SQLite 中，使用缩写时，offset 总是优先于 limit。紧随 limit 关键字的值是 offset 2。也要注意到，offset 依赖于 limit。也就是说，您可以只用 limit，不带 offset，凡是反过来都不行。

■注意：在操作管道中，limit 和 offset 是放在最后面的。人们常常误解 limit/offset 会加速查询，因为限制了 where 子句收集的行数目。事实并非如此，如果这样成立，那么 order by 就不能正常工作。因为 order by 是先将整个结果

排序才能获得正确的排序，但是性能会有细微改善，因为 SQLite 在处理中只需要追踪前 10 个最大的值，这个改善不是很大。

函数（Function)和聚合（Aggregate)

SQLite 提供了多种内置的函数和聚合，可以用在不同的子句中。函数的种类包括：数学函数，如计算绝对值的 abs()、字符串格式函数、将字符串的值转化为大写或小写的 upper()和 lower()。例如：

```
sqlite> select upper('hello newman'), length('hello newman'), abs(-12);

upper('hello newman')  length('hello newman')  abs(-12)
---------------------  ----------------------   ----------
HELLO NEWMAN           12                       12
```

函数名是不分大小写的（例如，upper()和 UPPER()是同一个函数）。函数可以接收字段值作为参数：

```
sqlite> select id, upper(name), length(name) from foods
        where type_id=1 limit 10;

id     upper(name)                 length(name)
-----  --------------------------- ------------
1      BAGELS                      6
2      BAGELS, RAISIN              14
3      BAVARIAN CREAM PIE          18
4      BEAR CLAWS                  10
5      BLACK AND WHITE COOKIES     23
6      BREAD (WITH NUTS)           17
7      BUTTERFINGERS               13
8      CARROT CAKE                 11
9      CHIPS AHOY COOKIES          18
10     CHOCOLATE BOBKA             15
```

因为函数可以是任意表达式的一部分，所以函数也可以用在 WHERE 子句中：

```
sqlite> select id, upper(name), length(name) from foods
        where length(name) < 5 limit 5;

id     upper(name)           length(name)
-----  --------------------  --------------------
```

```
36      PIE                     3
48      BRAN                    4
56      KIX                     3
57      LIFE                    4
80      DUCK                    4
```

67 聚合是一类特殊的函数，它从一组记录中计算聚合值。标准的聚合函数包括 sum()、avg()、count()、min()和 max()。例如，要得到烘烤食品（type_id=1）的数量，可使用如下语句：

```
sqlite> select count(*) from foods where type_id=1;

count
-----
47
```

count()聚合返回关系中所有行的数目。看到聚合，您就应该想到“对表中的每一行做某种运行”。

聚合不仅可以聚合字段，也可以聚合任何表达式，包括函数。例如，要获取所有食品名称的长度，可以在 length（name）上应用聚合函数 avg，如下所示：

```
sqlite> select avg(length(name)) from foods;

avg(length(name))
-----------------
12.58
```

聚合可以在 select 子句中操作。它们不仅可以对 from 子句选中行的值进行计算，也可以对 where 子句选中行的值进行计算。

虽然 SQLite 有一组标准的常用的 SQL 函数和聚合，但是还需要注意到，SQLite C API 允许您创建自定义的函数和聚合。相关内容将在第 7 章详细讨论。

分组（Grouping）

聚合的主要部分就是分组。也就是说，聚合不只是能够计算整个结果集的聚合值，还可以把结果集分成多个组，然后计算每个组的聚合值。这些都可以在一步中完成，方法就是使用 group by 子句。例如：

```
sqlite> select type_id from foods group by type_id;
```

```
type_id
----------
1
2
3
.
.
.
15
```

group by 与 select 有些不同，需要开动脑筋想象一下。如图 3-7 所示，演示了 group by 处理流程。从操作上讲，group by 介于 where 和 select 子句中间。group by 接收 where 的输出，并将其分隔成共享某个字段（或多个字段）上同等值的小组，这些组再传递给 select 子句。本例中，有 15 个不同的食品类型（type_id 取值从 1 到 15），因此，group by 将所有的行依据 type_id 不同分到 15 个组中。select 接收所有的组，提取共有的 type_id，将其放到单独的行。因此，结果中有 15 行，每小组只占一行。

68

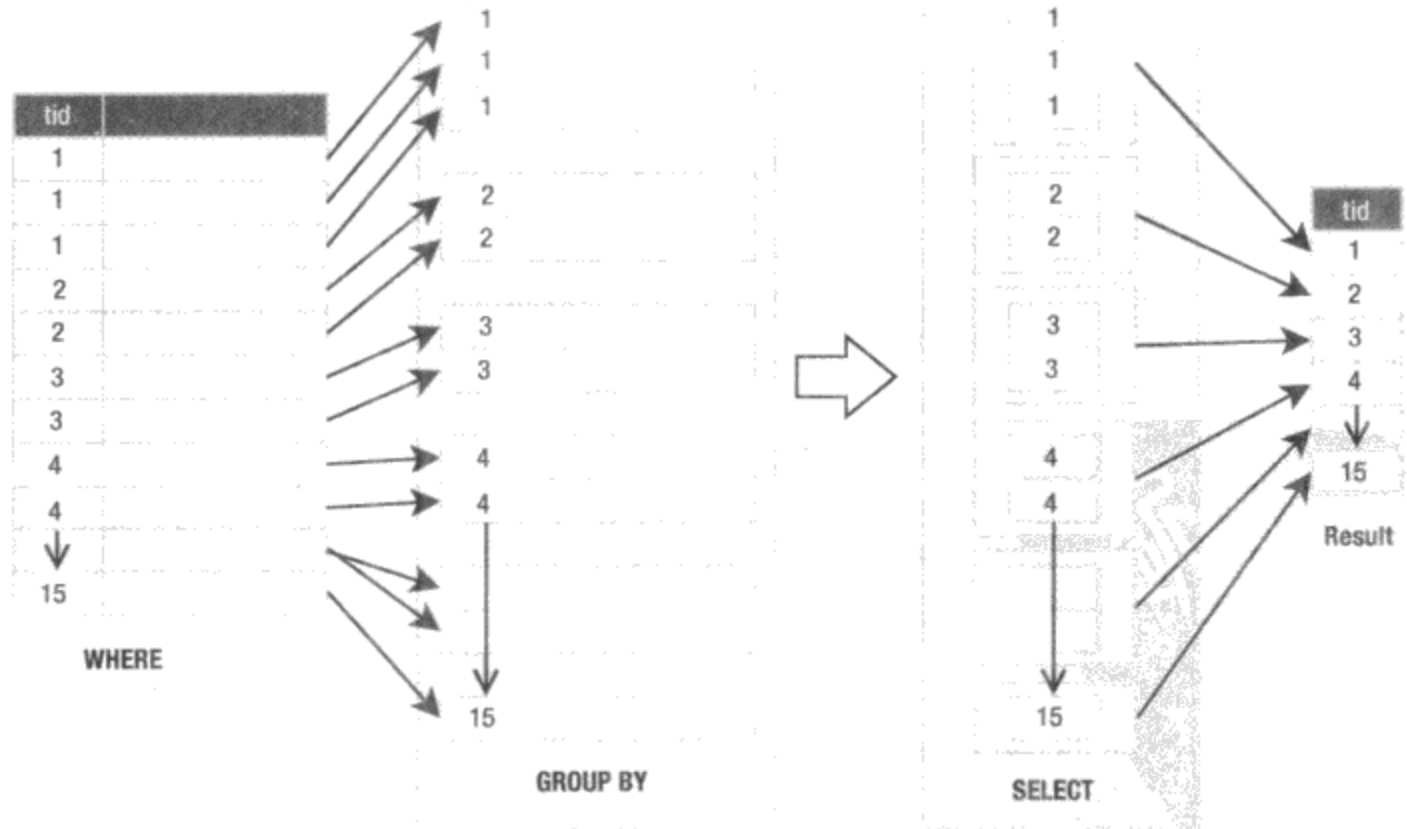


图 3-7　group by 处理流程

使用 group by 时，select 子句对每组单独应用聚合，而不是对整个结果进行聚合。

因此，聚合对每组生成一个值，并将这些组的行作为单行。例如，如果对前面的例子应用 count 聚合，获取每个 type_id 组的记录数目：

```
sqlite> select type_id, count(*) from foods group by type_id;
type_id     count(*)
----------  ----------
1           47
2           15
3           23
4           22
5           17
6           4
7           60
8           23
9           61
10          36
11          16
12          23
13          14
14          19
15          32
```

69

如图 3-8 所示，聚合 count()运行了 15 次——每一组运行一次。注意，图中没有完整地展示每组实际的所有记录（也就是说，type_id=1 组没有显示全部的 47 个记录）。

70 type_id=1（烘烤食品）的记录总数是 47 个，type_id=2（谷物类食品）的数目是 15 个，type_id=2（鸡/家禽食品）的记录数为 23 个，如此等等，您可以执行如下查询获得这些信息：

```
select count(*) from foods where type_id=1;
select count(*) from foods where type_id=2;
select count(*) from foods where type_id=3;
.
.
.
select count(*) from foods where type_id=15;
```

或者使用如下的 select 和 group by 一次获得结果。

```
select type_id, count(*) from foods group by type_id;
```

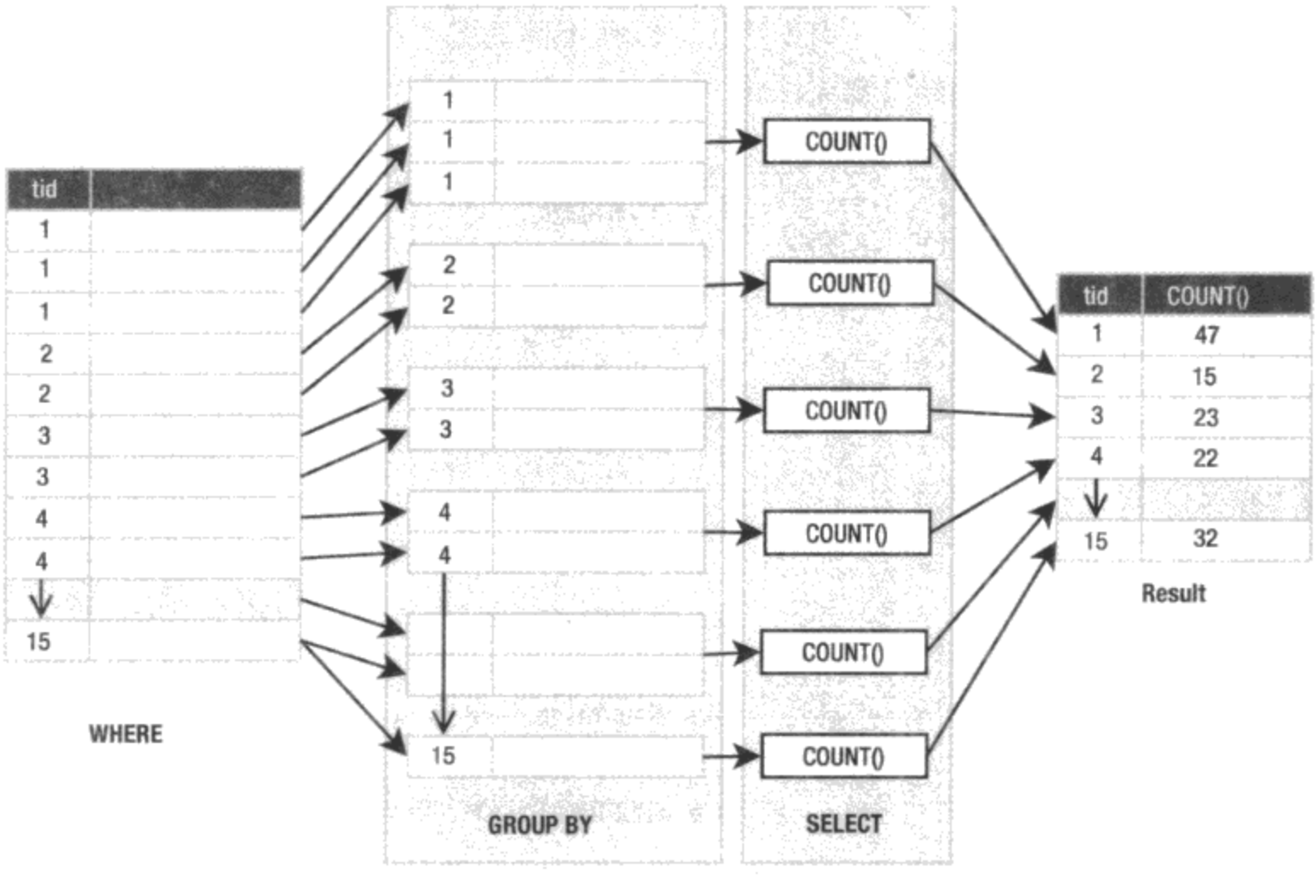


图 3-8　group by 与聚合

group by 使用类似的值创建分组，但没有在 select 子句处理前过滤这些组。having 具备这一功能，having 是一个可以应用到 group by 的断言。它从 groupby 中过滤组的方式与 where 子句从 from 子句中过滤行的方式相同。唯一不同的是，where 子句的预测是针对单个行的，而 having 的断言是针对聚合值的。

参考前面的例子，这次只关注那些食品小于 20 个的食品类型。

```
sqlite> select type_id, count(*) from foods
        group by type_id having count(*) < 20;
type_id      count(*)
----------   ----------
2            15
5            17
6            4
11           16
13           14
14           19
```

71 断言 count(*)<20 应用到所有的组上。任何不满足这一条件（条件是所在组食物数目为 20 或者大于 20）的结果都无法传递到 select 子句。图 3-9 就说明了这一限制。

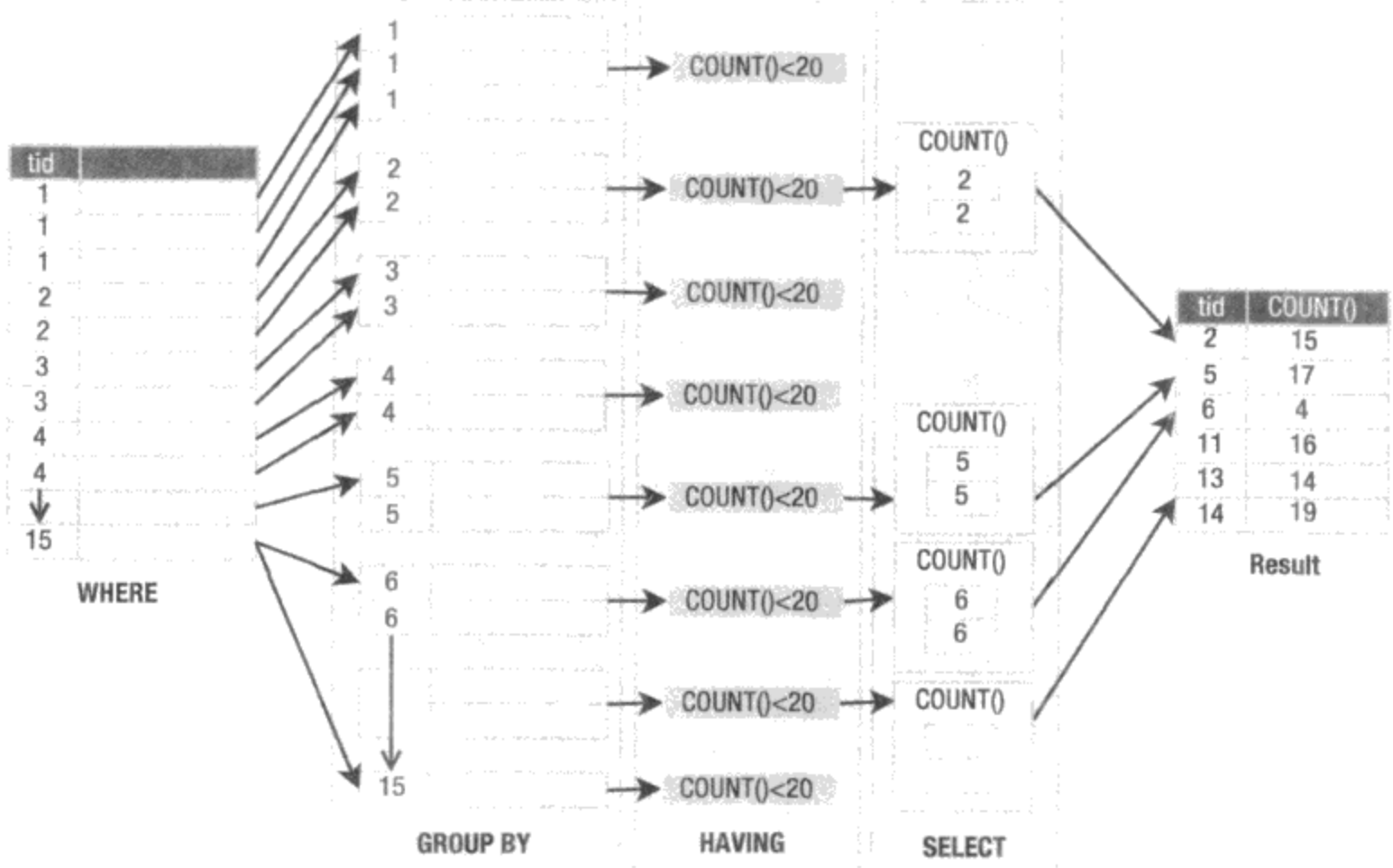


图 3-9 组限制 having

第三列显示了按照 type_id 排序的组，同时显示了 type_id 的值。实际显示值有所不同，本例主要用于说明，并非真实数据，每组只显示两行作为代表。

group by 和 having 一起工作可以对 group by 有约束。group by 接收 where 子句的约束，将结果行分成共享某值的组。having 对每组应用过滤，通过过滤的组传递给 select 子句来做聚合和映射。

72 ■注意：包括 SQLite 在内的一些数据库允许构建聚合和非聚合字段的 select 语句，非聚合字段就是 select 中没有分组的字段。例如，SQLite 允许执行如下 SQL 语句：

```
select type_id, count(*) from foods
```

因为聚合（count）计算整个输入表，这样会跳过所有的过滤步骤，造成返回的行数不匹配。本例中的 count 返回一行，但是没有让 SQLite 对 type_id 进行分组。然而，SQLite 会返回一些内容。不幸的是，这种语句的结果是无意义的——不要依赖于那些没有分组的非聚合字段的 SQL 语句，因为它们的结果是不确定的值。

去掉重复

distinct 处理 select 的结果并过滤掉其中重复的行。例如，您想从 foods 表中取得所有不同的 type_id 值：

```
sqlite> select distinct type_id from foods;

type_id
----------
1
2
3
.
.
.
15
```

该语句按照如下流程工作：where 子句返回表 foods 的所有行（共 412 行）。select 子句取出其中的 type_id 字段，最终 distinct 删除重复行，从 412 行减少到 15 行，所有的值都是唯一的。

多表连接

连接（join）是多表（关系）数据工作的关键，它是 select 命令的第一个操作。连接操作的结果作为输入，供 select 语句的其他部分（过滤）处理。

通过例子理解 SQLite 是最好的途径。foods 表有一个字段是 type_id，该字段的值都与 food_types 表中的 id 字段对应。这样两个表之间就存在关联了，即 foods.type_id 的任何值都必须是 food_types.id 中存在的。id 是表 food_types 的主键（后面会解释），

foods.type_id 由于这种关系，称为外键，它包含（引用）另一个表的主键，这种关系称为外键关系。

73 使用这种关系，基于这两个字段将表 foods 和表 food_type 连接起来形成新关系，这样可以提供更详细的信息，也就是为 foods 表的每个食品提供 food_type.name，如下所示：

```
sqlite> select foods.name, food_types.name
        from foods, food_types
        where foods.type_id=food_types.id limit 10;

name                        name
--------------------------- ---------------
Bagels                      Bakery
Bagels, raisin              Bakery
Bavarian Cream Pie          Bakery
Bear Claws                  Bakery
Black and White cookies     Bakery
Bread (with nuts)           Bakery
Butterfingers               Bakery
Carrot Cake                 Bakery
Chips Ahoy Cookies          Bakery
```

可以看到：foods.name 在结果的第一列，紧接着是 food_types.name。前面的每一行都关联到后面使用的 foods.type_id→food_types.id 关系（见图 3-10）。

注意：本例中，我们使用新的方法在 select 中指定字段，而不是仅仅指定字段名称。我们使用了 table_name.column_name 这种方式，因为在 select 语句中有多个表，如果字段名称在所有的表中是唯一的，数据库可以区分出字段属于哪个表。如果某个字段名称也在其他连接的表中存在，数据库无法判断您在引用哪个表的字段，将返回错误。实际中，连接表时，使用 table_name.column_name 是一种非常好的避免歧义的方式。相关内容在“命名与别名”一节会详细讨论。

要实现连接，数据库需要找出匹配的行。对于第一个表中的每一行，数据库都要查询第二个表的所有行，寻找那些连接字段具有相同值的行，然后将它们包含到输入关系中。本例中，通过连接两个表的行构建一个符合关系。

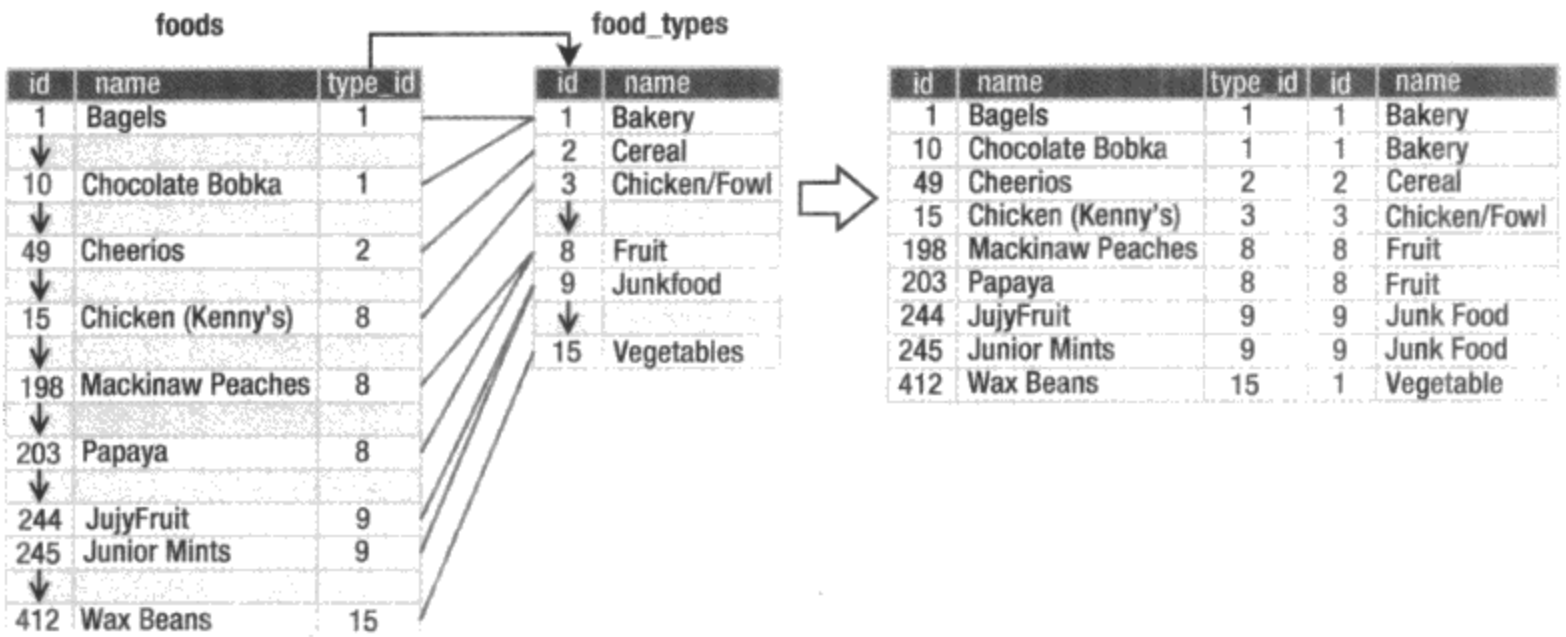


图 3-10　foods 和 food_types 连接

子查询操作（where、group by 等）与主查询的工作方式相同，只是通过连接表改变了输入。因此，SQLite 支持六种不同类型的连接，其中一种称为内连接的连接，是最普遍的。

外连接

内连接就是通过表中的两个字段进行连接，如前面例子所示。它也是最常见（也许也是最有用的）的连接类型。

内连接使用关系代数的另一种集合操作，称为交叉，即找出同时存在于两个集合的元素。如图 3-11 所示，集合{1, 2, 8, 9}和集合{1, 3, 5, 8}的交集是{1, 8}。交叉操作可以通过维恩图展示。

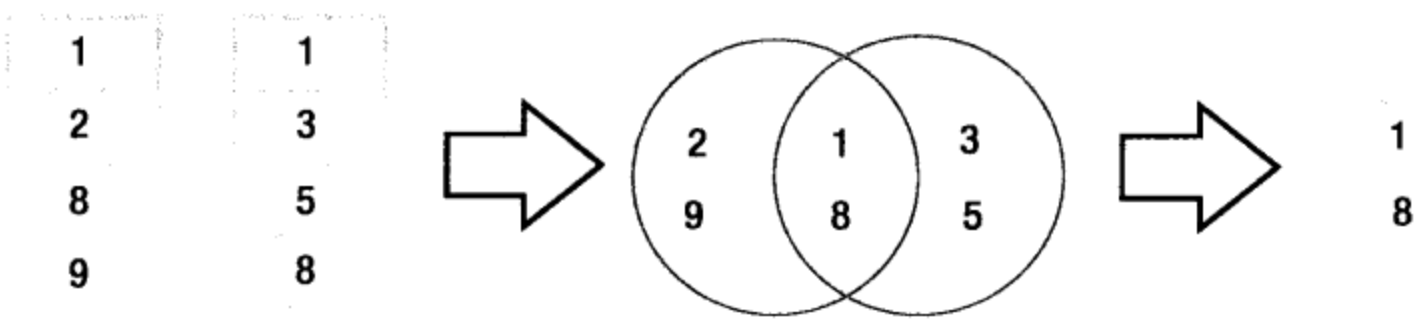


图 3-11　交集

该图准确地展示了内连接是如何工作的，但是连接中的集合通常在相关字段上具有

共同元素。假设图 3-11 中左边集合代表 foods.type_id 字段，右边集合代表 food_types.id 字段。考虑一下匹配字段，内连接将从两边集合中找出那些包含相似值的行，然后结合两者组成结果集（见图 3-12）。注意，本例假设所展示的记录是 foods 和 food_types 表仅有的记录。

75

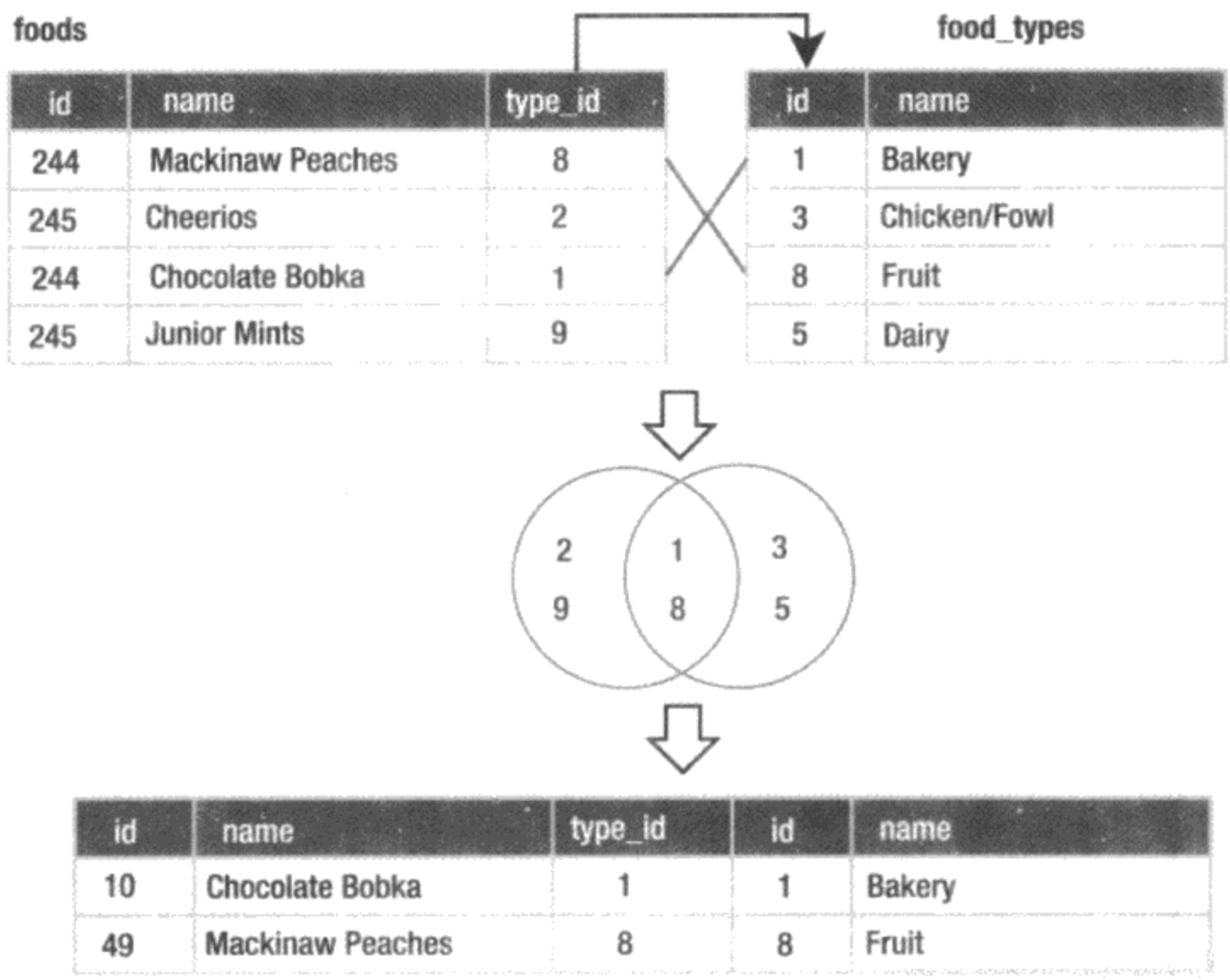


图 3-12　food 的交集连接

内连接只返回满足给定字段关系的行，也称为连接条件。它们回答问题“B 中哪些行匹配 A 中的给定关系？”下面我们用实际的语句展示一下这种 SQL。

```
select *
From foods inner join food_types on foods.id = food_types.id
```

只要看见实际结果，很容易理解语法。

交叉连接

想象一下，如果没有连接条件，您会得到什么？本例中，如果两个表没有通过任何方式关联，select 会产生一种更基础的连接（也是最根本的），称为交叉连接、笛卡儿积或者交叉乘积。笛卡儿积是基本的关系运算之一。它是强制的几乎无意义的连接，即使第一个表中的所有行和第二个表中的所有行联合起来。

在 SQL 中，交叉连接 foods 和 food_types 的表达式如下：

```
select * from foods, food_types;
```

from 在缺乏其他条件时产生交叉连接，结果如下。foods 中的每一行都与 food_types 的所有行组合在一起，但是关键的是，不是通过关联 id 和 type_id 值。交叉连接中，行之间不存在关系，也没有连接条件，但是它们只是简单地组合在一起。

76

```
1       1       Bagels      1           Bakery
1       1       Bagels      2           Cereal
1       1       Bagels      3           Chicken/Fowl
...
(此处省略 6174 行)
...
412     15      Wax Beans   13          Seafood
412     15      Wax Beans   14          Soup
412     15      Wax Beans   15          Vegetables
```

您可能会问，交叉连接有什么用途。坦白地说，这也是在使用这种逻辑前您必须问自己的问题。您真的想让一个表中的行跨到另一表中的每一行吗？答案通常是 no。

外连接

四种连接中的其余三种都是外连接。内连接根据给定关系选择表中的行。外连接选择内连接的所有行外加一些关系之外的行。三种外连接类型分别为左外连接、右外连接和全外连接。左外连接操作 SQL 命令中的“左表”，例如：

```
select *
from foods left outer join foods_episodes on foods.id=foods_episodes.food_id;
```

foods 是其中的左表，它是左外连接中很重要的表。左外连接试图将 foods 中的所有行与 foods_episodes 的所有行进行连接关系（foods.id=foods_episodes.food_id）的匹配，所有匹配的行都包含在结果集中。但是，如果我们在 foods 表中注册一些食品，这些食

品没有在 foods_episodes 中出现，foods 表中没有匹配 foods_episodes 的剩余行仍然会出现在结果集中，foods_episodes 没有提供相应的行，它会以 null 补充。

右外连接的工作方式类似，不管是否匹配，右表的行都包含在结果集中。从操作上讲，左外连接和右外连接是相同的，它们做同样的事情，只是在顺序和语法上有所不同。作为 SQLite 用户，这意味着任何通过右外连接的解决方法都可以用同样的左外连接解决。

全外连接是左外连接和右外连接的结合。它包含所有的匹配行，然后是右边和左边表的不匹配行。目前，SQLite 不支持右外连接和全外连接，但是正如前面提到的，右外连接可以用左外连接代替，全外连接可以通过使用复合查询执行（参考本章后面的“复合查询”）。

77

自然连接

关系操作中介绍的最后一个就是自然连接。它实际上是内连接的另一种形式，但是有些语法和习惯上的改动。自然连接通过表中共有的字段名称将两个表连接起来。因此，使用自然连接时，不用添加连接条件就可以获得内连接的结果。

自然连接会连接两个表中所有具有相同名称的字段。向表中添加或从中删除一个字段可以极大地影响自然连接查询的结果，产生不可预测的结果，特别是如果您的表设计一直改变的话，最好是清晰地定义查询中的连接条件，而不是依赖于表模式的语义关系。

语法偏好

从语法上讲，可以通过各种方式指定连接。foods 和 food_types 的内连接可以通过 where 子句隐含实现。

```
select * from foods, food_types where foods.id=food_types.food_id;
```

当列出数据库的多个表时，默认会做连接操作——最低限度也会做交叉连接。本例中的 where 是内连接。

这种隐式形式虽然很简洁，但实际上是一种应该避免的过时语法形式。SQL 中正确表达连接的方法是（SQL92 中规定的）使用连接关键字。一般形式如下：

```
select heading from left_table join_type right_table on join_condition;
```

这种显式形式可以用在所有的连接中，例如：

```
select * from foods inner join food_types on foods.id=food_types.food_id;
select * from foods left outer join food_types on foods.id=food_types.food_id;
select * from foods cross join food_types;
```

使用 ANSI 连接语法最重要的原因是有些查询类型只能使用连接关键词语法完成。特别是外连接——左外连接、右外连接和全外连接。

名称和别名

连接表时，如果两个表有同样名称的字段，就会产生歧义。例如，前面连接表时使用了 selectid，那么 SQLite 应该返回哪个表的 id 呢？（表 food_types 和 foods 都有名称为 id 的字段）为了解决这类问题，您可以在字段名前面加上表名以消除歧义。

另一种有用的方法是别名。如果您的表名较长，并且不想每次都在字段名前面加上表名，您可以使用别名。别名实际上是一种称为重命名的关系操作，重命名操作可以简单地将一个关系命名成另一个关系。例如：

```
select foods.name, food_types.name
from foods, food_types
where foods.type_id = food_types.id
limit 10;
```


这样输入就比较方便。可以在 from 子句中对表重命名，然后在其余地方直接使用，如下例所示：

```
select f.name, t.name
from foods f, food_types t
where f.type_id = t.id
limit 10;
```

其中，foods 表的别名为 f，foods_types 表的别名为 t，其他对表 foods 或 foods_types 的引用都可以使用别名 f 和 t。别名使自我连接（表与自身进行连接）成为可能。例如，如果想找出哪些 season 等于 4 的食品在其他 season 中提及过，首先您必须获取 episodes 和 season 等于 4 的食品清单，这个可以通过连接 episodes 和 episodes_foods 表实现，然后找出类似的 season 不等于 4 的食品清单，最后对这两个清单进行联合。下面展示自我连接的例子：

```
select f.name as food, e1.name, e1.season, e2.name, e2.season
from  episodes e1, foods_episodes fe1, foods f,
      episodes e2, foods_episodes fe2
where
  -- Get foods in season 4
  (e1.id = fe1.episode_id and e1.season = 4) and fe1.food_id = f.id
  -- Link foods with all other episodes
  and (fe1.food_id = fe2.food_id)
  -- Link with their respective episodes and filter out e1's season
  and (fe2.episode_id = e2.id AND e2.season != e1.season)
order by f.name;
```

food	name	season	name	season
Bouillabaisse	The Shoes	4	The Stake Out	1
Decaf Cappuccino	The Pitch	4	The Good Samaritan	3
Decaf Cappuccino	The Ticket	4	The Good Samaritan	3
Egg Salad	The Trip 1	4	Male Unbonding	1
Egg Salad	The Trip 1	4	The Stock Tip	1
Mints	The Trip 1	4	The Cartoon	9
Snapple	The Virgin	4	The Abstinence	8
Tic Tacs	The Trip 1	4	The Merv Griffin Show	9
Tic Tacs	The Contest	4	The Merv Griffin Show	9
Tuna	The Trip 1	4	The Stall	5
Turkey Club	The Bubble Boy	4	The Soup	6
Turkey Club	The Bubble Boy	4	The Wizard	9

这里解释一下 SQL 是如何执行的。本例在 where 子句中使用两个自我连接，有两个 episodes 和 foods_episodes 的实例，在此把它们当成两个独立的表。查询连接 foods_episodes 表到自身和 episodes 两个实例上。两个 episodes 实例通过不等于条件彼此关联，以确保它们在不同的 season 中。可以用同样的方式对字段和表达式使用别名。SQLite 中别名的一般语法如下：

79

```
Select base-name [[as] alias] …
```

其中，as 关键字是可选的，但是很多人倾向于保留 as，这样别名显得更清晰，就不会将别名与基本的字段名和表达式混淆。

子查询

子查询是指 select 语句中又嵌套 select 语句。子查询可以应用在很多方面，它们可以在一般表达式不能使用的地方工作，也可以应用在 select 语句的各个位置。子查询在其他语句中也有用途。

也许，子查询最常应用的地方是 where 子句，特别是在 in 操作符中。in 操作符是一个双目操作，输入一个值和一列值，如果输入的单个值存在于列值中，返回真，否则为假。例如：

```
sqlite> select 1 in (1,2,3);
1
sqlite> select 2 in (3,4,5);
0
sqlite> select count(*)
   ...> from foods
   ...> where type_id in (1,2);
62
```

使用子查询可以重写上面的查询：

```
sqlite> select count(*)
   ...> from foods
   ...> where type_id in
   ...>  (select id
   ...>   from food_types
   ...>   where name='Bakery' or name='Cereal');
62
```

select 子句中的子查询可以用来从其他表向结果集添加额外数据。例如，要查询每种系列食品的数目，对食品系列的数据数目查询可以在 select 子句的子查询中执行。

```
sqlite> select name,
        (select count(id) from foods_episodes where food_id=f.id) count
        from foods f order by count desc limit 10;

name             count
----------       ----------
Hot Dog          5
Pizza            4
Ketchup          4
```

80
```
Kasha             4
Shrimp            3
Lobster           3
Turkey Sandwich   3
Turkey Club       3
Egg Salad         3
Tic Tacs          3
```

order by 和 limit 子句用来创建前 10 项，其中 Hot Dog 处于第一位。注意到在封闭的 select 命令中子查询引用表 food_id=f.id。变量 f.id 存在于外部查询。本例中的子查询也称为相关子查询，因为它引用或关联外部查询的变量。

在子查询中可以使用 order by 子句。下面的 SQL 语句根据各个食品所属组的食品数量由大到小排序。

```
select * from foods f
order by (select count(type_id)
from foods where type_id=f.type_id) desc;
```

本例中的 order by 没有引用结果中的任何指定字段。它如何工作呢？order by 子查询在每行上都运行，并将结果关联到指定行。可以将其当成结果集中不可见的字段，它是使用该字段排序的。

最后看看 from 子句，有时可能想与其他结果进行连接，而不是单一的基本表。这也是子查询的功能。

```
select f.name, types.name from foods f
inner join (select * from food_types where id=6) types
on f.type_id=types.id;
```

```
name                         name
---------------------------  -----
Generic (as a meal)          Dip
Good Dip                     Dip
Guacamole Dip                Dip
Hummus                       Dip
```

注意，在 from 子句中使用子查询需要重命名操作。本例中，子查询重命名为 types。from 子句中的子查询作为一个关系源常常被称为内联视图或派生表。

记住，子查询可以用在关系表达式的任何地方。掌握如何、在哪里以及什么时候使用子查询的最好方法是不断地使用它们看看能得到什么。SQL 中解决问题的方法往往不

止一种。理解了重点，您可以做出更明智的关于重写某个查询是否能获得更高效率的决定。

复合查询

复合查询与子查询相反，它是使用三种特殊的关系操作符（联合、交叉连接和差集）处理多个查询的结果。在 SQLite 中对应使用关键字 union、intersect 和 except。

复合查询操作需要如下一些参数：

- 涉及的关系的字段数目必须相同。
- 只能有一个 order by 子句，并且处在复合查询的最末尾，对联合结果进行排序。

此外，复合查询中的关系从左向右处理。

union 操作输入两个关系：A 和 B，将两者联合成一个只包含 A 和 B 中非重复字段的单一关系。SQL 中，union 联合两个 select 语句的结果。默认情况下，union 会消除重复数据。如果想在结果中保留重复数据，可以使用 union all。例如，下面的 SQL 语句是要找出 foods 表中最高频率和最低频率的食品：

```
select f.*, top_foods.count from foods f
inner join
  (select food_id, count(food_id) as count from foods_episodes
     group by food_id
     order by count(food_id) desc limit 1) top_foods
  on f.id=top_foods.food_id
union
select f.*, bottom_foods.count from foods f
inner join
  (select food_id, count(food_id) as count from foods_episodes
     group by food_id
     order by count(food_id) limit 1) bottom_foods
  on f.id=bottom_foods.food_id
order by top_foods.count desc;
```

```
id          type_id     name        top_foods.count
----------  ----------  ----------  ---------------
288         10          Hot Dog     5
1           1           Bagels      1
```

两个查询值返回一行，它们之间的唯一差别是排序方式的不同。union 简单地将两

行联合成一个关系。

intersect 操作输入两个关系 A 和 B，选择那些既在 A 也在 B 的行。下面的 SQL 语句是使用 intersect 找出 episodes 介于 3～5 之间的处于前 10 位的食品。

```
select f.* from foods f
inner join
  (select food_id, count(food_id) as count
     from foods_episodes
     group by food_id
     order by count(food_id) desc limit 10) top_foods
on f.id=top_foods.food_id
intersect
select f.* from foods f
  inner join foods_episodes fe on f.id = fe.food_id
  inner join episodes e on fe.episode_id = e.id
  where e.season between 3 and 5
order by f.name;
```

```
id          type_id     name
----------  ----------  --------------------
4           1           Bear Claws
146         7           Decaf Cappuccino
153         7           Hennigen's
55          2           Kasha
94          4           Ketchup
164         7           Naya Water
317         11          Pizza
```

要产生处于前 10 位的食品，需要在第一个 select 语句中排序（order by）。因为复合查询只允许在语句末尾有一个 order by，通过对计算前 10 位食品的子查询执行内连接绕过这一限制，子查询可以有 order by，因为它独立于复合查询的运行，内连接会产生包含前 10 位食品的关系。第二个查询返回 episodes 介于 3～5 之间的食品。intersect 操作找出所有匹配的行。

except 操作输入两个关系 A 和 B，找出所有在 A 但不在 B 的行。通过将前面例子中的 intersect 改为 except，可以找出 episodes 介于 3～5 之间的不在前 10 位的食品：

```
select f.* from foods f
inner join
  (select food_id, count(food_id) as count from foods_episodes
```

```
    group by food_id
    order by count(food_id) desc limit 10) top_foods
  on f.id=top_foods.food_id
except
select f.* from foods f
  inner join foods_episodes fe on f.id = fe.food_id
  inner join episodes e on fe.episode_id = e.id
  where e.season between 3 and 5
order by f.name;
```

```
id          type_id     name
----------  ----------  --------
192         8           Banana
133         7           Bosco
288         10          Hot Dog
```

正如前提到的，except 操作也就是 SQL 中关系代数的差集（difference）操作。

需要以不同方式处理类似的集合时，复合查询非常有用。基本上，不能在单一 select 语句中表达的任何查询，都可以使用复合查询从一个 select 中获取一部分或其他部分（也许需要更多的 select），然后相应地处理它们。

83

条件结果

case 表达式允许在 select 语句中处理各种情况，它有两种形式。第一种是最简单的，即接收静态值并列出各种情况下的 case 返回值：

```
case value
  when x then value_x
  when y then value_y
  when z then value_z
  else default_value
end
```

简单示例：

```
select name || case type_id
                when 7 then ' is a drink'
                when 8 then ' is a fruit'
                when 9 then ' is junkfood'
                when 13 then ' is seafood'
                else null
```

```
                . end description
from foods
where description is not null
order by name
limit 10;
```

```
description
-----------------------------------------------
All Day Sucker is junkfood
Almond Joy is junkfood
Apple is a fruit
Apple Cider is a drink
Apple Pie is a fruit
Arabian Mocha Java (beans) is a drink
Avocado is a fruit
Banana is a fruit
Beaujolais is a drink
```

本例中的 case 表达式处理不同类型的 type_id 值，返回恰当的字符串，返回值称为 description，在 end 关键字之后，通过字符串连接符（||）和 name 拼接在一起，组成一个完整的句子。对于 when 中没有指定的所有 type_ids，case 返回 null。select 语句在 where 语句中过滤了 null，因此，所有的返回值都在 case 表达式处理情况中。

第二种 case 形式允许 when 条件中有表达式。形式如下：

```
case
  when condition1 then value1
  when condition2 then value2
  when condition3 then value3
  else default_value
end
```

case 在子选择语句中与聚合工作方式相同。下面的 SQL 语句从 foods 表中提取频率。

```
select name,(select
               case
                  when count(*) > 4 then 'Very High'
                  when count(*) = 4 then 'High'
                  when count(*) in (2,3) then 'Moderate'
                  else 'Low'
               end
             from foods_episodes
```

```
            where food_id=f.id) frequency
from foods f
where frequency like '%High'
```

```
name        frequency
---------   ----------
Kasha       High
Ketchup     High
Hot Dog     Very High
Pizza       High
```

该查询对每行执行子查询，子查询按照所属食品系列的食品数量对其进行分类，查询的结果包含在称为 frequency 的字段中。where 断言过滤含有 high 的 frequency 字段值。

case 表达式中只执行一个条件。即是说，如果满足的条件超过一个，也只执行第一个；如果没有满足条件且没有定义 else 条件，case 返回 null。

处理 SQLite 中的 Null

绝大多数关系数据库都通过一种称为 null 特殊占位符支持“未知”或“不可知”的概念，null 是缺失信息的占位符，本身不是值。null 表示该位置没有值：null 不是什么其他值，Null 不是真，也不是假，不是零，也不是空字符串。简单地理解，null 就是自己：null。可能一些人不明白其中的含义。掌握 SQLite 中的 null，需要了解一些关键规则和观点。

85

第一，为了在逻辑表达式中使用 null，SQLite 使用所谓的三值（或三态）逻辑，null 是真假值之一。表 3-2 显示了逻辑与（and）和逻辑或（or）与 null 的关系。

表 3-2　与 null 相关的逻辑与和逻辑或

x	y	X AND y	x OR y
True	True	True	True
True	False	False	True
True	NULL	NULL	True
False	False	False	False
False	NULL	False	NULL
NULL	NULL	NULL	NULL

您可以试一些简单的 select 语句看看 null 的效果。

第二，可以通过 is null 或者 is not null 操作符检测 null 是否存在。如果试图使用其他操作符，例如 equal、greater than 等，您可能会惊呆，看看第三条关于 null 的规则。

第三条也是最后一条，记住 null 不等于其他任何值，包括 null，不可以将 null 与其他值进行比较，null 绝不会大于或小于其他任何 null。像下面这样粗心的语句常常不能返回任何行。

```
select *
from mytable
where myvalue = null;
```

null 不等于任何值，所以该 SQL 语句不会返回任何值。现在已经掌握了基本知识，应该感到庆幸的是，SQLite 提供了一些与 null 相关的额外功能。其中第一个功能就是对“null 不等于任何值”这一限制进行变通。从 SQLite v 3.6.19 开始，is 操作符可以让一个 null 等于另一个 null。最基本的形式是可以执行简单查询 SQLite，看看 NULL 是否是 NULL。

```
sqlite> select NULL is NULL;
1
```

前面已经提到，非零的任何值都表示“真”。该例中，SQLite 告诉我们 NULL 等于 NULL。但是不要依赖这种行为。SQL 的三态逻辑可能有些不适用，如果您期望在其他系统和编程语言中与 is 操作符一起工作的三态逻辑出现这样的预期结果，可能会遇到问题，这是公认的规则。

86

coalesce 函数作为 SQL99 标准的一部分，将一组值作为输入并返回其中第一个非 null。如下例所示：

```
select coalesce(null, 7, null, 4)
```

在本例中，coalesce 将返回第一个非 null 值 7。这在执行一些算术运算时是有帮助的，可以检测是否有 null 返回，如果有，可以返回更有意义的值（例如 0）来代替。

相反，nullif 函数输入两个参数，如果两个参数值相同，则返回 null；否则，返回第一个参数：

```
sqlite> select nullif(1,1);
null
sqlite> select nullif(1,2);
1
```

如果使用 null，需要特别注意断言和聚合中引用可能存在 null 值字段的查询，否则，null 可能在聚合中产生意想不到的效果。

总结

祝贺您，到这里您已经学习完了 SQLite 中的 select 命令。您不仅知道了该命令如何工作，还了解了其中的关系理论。现在已经熟悉如何使用 select 语句对数据进行查询、连接、聚合、汇总以及仔细分析了。

下一章继续讨论 SQL，我们在 select 基础上介绍 DML 的其他成员以及 DDL 和 SQLite 中其他有用的 SQL 指令。

第 4 章

SQLite 中的高级 SQL

第 3 章介绍了 select 语句，本章介绍 SQL 语言的其他功能，内容涵盖了 SQLite 实现的其余 SQL，包括修改数据的 insert、update 和 delete，保护数据的约束，以及更高级的主题，例如表的创建和数据类型。

修改数据

与 select 命令相比，用于修改数据的语句很容易理解和使用。有 3 个 DML 语句用于修改数据——insert、update 和 delete，不过它们远比字面含义看起来做的事情要多。

插入记录

使用 insert 命令向表中插入记录。insert 在单表上工作，使用 insert 命令可以一次插入一条记录，使用 select 命令可以一次插入多条记录。insert 语句的一般格式为：

```
insert into table (column_list) values (value_list);
```

变量 table 指明数据插入到哪个表中。变量 column_list 是用逗号分隔的字段名称，这些字段必须是表中存在的。变量 value_list 是用逗号分隔的值列表，这些值与 column_list 中的字段一一对应。

插入一行

例如，下列语句向 foods 表插入数据：

```
sqlite> insert into foods (name, type_id) values ('Cinnamon Bobka', 1);
```

该语句插入一行，指定两列值'Cinnamon Bobka'——第一个值，对应字段列表中的第一个字段 name。类似地，1 对应 type_id，就是第二个字段。注意，其中没有 id 字段。本例中，数据库使用默认值。因为 id 是整型主键，它会自动产生相关的新值（后面的“主键约束”中会讲到）。可以用简单的 select 命令验证一下，也可以查询到 id 值。


```
sqlite> select * from foods where name='Cinnamon Bobka';

id          type_id     name
----------  ----------  --------------
413         1           Cinnamon Bobka

sqlite> select max(id) from foods;

MAX(id)
----------
413

sqlite> select last_insert_rowid();

last_insert_rowid()
-------------------
413
```

注意：值 413 是为 id 字段自动产生的，它也是该字段的最大值。SQLite 提供单增长值，您可以使用 SQLite 内建的 last_insert_rowid()验证这一点，该函数返回最后一个自增长值，如示例所示。

如果在 insert 语句中为表的所有字段提供值，可以省略字段列表。本例中，假定数据库值的顺序与字段在 create table 语句的顺序一一对应，例如：

```
sqlite> insert into foods values(NULL, 1, 'Blueberry Bobka');
sqlite> select * from foods where name like '%Bobka';
id          type_id     name
----------  ----------  ---------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
414         1           Blueberry Bobka
```

注意参数的顺序，'Blueberry Bobka' 在 1 的后面，这是因为表创建的方式如下。（若要查看表的定义，可在 shell 提示符下输入.schema foods。）

```
sqlite> .schema foods
CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
```

第一个字段是 id，紧接着是 type_id，最后是 name。因此，这也是 insert 语句中的值列表顺序。为什么在 insert 语句中为 id 赋值 NULL？SQLite 知道 foods 表中的 id 字段是自增长字段，并指定 null 表示不提供值。不指定值会自动触发自增长值的产生，这样做是为了方便，没有更深的含义或者理论。本章后面会讨论自增长字段的内容。

89

插入一组行

子查询可以用在 insert 语句中，既可以作为值列表的一部分，也可以完全替代整个值列表。指定子查询为值列表时，实际上是在插入一组行，因为子查询返回的通常是一组行。下面的例子只产生一行值：

```
insert into foods
values (null,
       (select id from food_types where name='Bakery'),
       'Blackberry Bobka');
select * from foods where name like '%Bobka';
```

```
id          type_id     name
----------  ----------  ---------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
414         1           Blueberry Bobka
415         1           Blackberry Bobka
```

这里不用硬编码输入 type_id，而是用 SQLite 来查询，例如：

```
insert into foods
select last_insert_rowid()+1, type_id, name from foods
where name='Chocolate Bobka';
select * from foods where name like '%Bobka';
```

```
id          type_id     name
----------  ----------  ---------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
```

```
414         1           Blueberry Bobka
415         1           Blackberry Bobks
416         1           Chocolate Bobka
```

本例使用 select 语句完全替代了值列表。只要 select 子句的字段数目与要插入的表的字段数目匹配，或者与提供的字段列表匹配，insert 语句就可以正常工作。上面的例子中，添加另一种 chocoklate bobka，使用 last_insert_rowid()+1 作为 id 字段的值，可以使用 NULL 代替。事实上，应该使用 NULL，而不是 last_insert_rowid()，因为如果之前会话中没有插入行，last_insert_rowid()会返回 0。

本例中的 last_insert_rowid()可以正常工作，但是最好不要在程序中这样做。

插入多行

使用 select 形式的 insert 可以一次插入多行。只要字段匹配，insert 可以插入结果集的所有行，例如：

```
sqlite> create table foods2 (id int, type_id int, name text);
sqlite> insert into foods2 select * from foods;
sqlite> select count(*) from foods2;
count(*)
--------------------
418
```

上述语句表示创建一个新表 foods2，并插入表 foods 的所有记录。

然而，有一种更方便的方式创建和插入表。create table 语句有特殊的语法，可以直接指定从 select 语句中创建表。前面的例子可以用下面的语法一步完成：

```
sqlite> create table foods2 as select * from foods;
sqlite> select count(*) from list;

count(*)
--------
418
```

create table 与从 foods 表选择数据插入表两步并为一步。这对于创建临时表特别有用。

```
create temp table list as
select f.name food, t.name name,
      (select count(episode_id)
```

```
    from foods_episodes where food_id=f.id) episodes
from foods f, food_types t
where f.type_id=t.id;
select * from list;
```

Food	Name	Episodes
Bagels	Bakery	1
Bagels, raisin	Bakery	2
Bavarian Cream Pie	Bakery	1
Bear Claws	Bakery	3
Black and White cook	Bakery	2
Bread (with nuts)	Bakery	1
Butterfingers	Bakery	1
Carrot Cake	Bakery	1
Chips Ahoy Cookies	Bakery	1
Chocolate Bobka	Bakery	1

使用 create table 的这种形式时，要意识到源表的任何约束是否定义在新表中。特别是，自增长字段不会在新表中创建，索引也不会创建，UNIOUE 约束等都不会创建。许多其他数据库称这种功能为 CTAS（即 Create Table As Select 首字母的组合），但是 CTAS 不是 SQLite 用户普遍使用的称呼。

有一点值得注意，插入行时的 unique 约束。也就是说，如果在定义为 unique 的字段中插入重复值，SQLite 会停止并报错。

```
sqlite> select max(id) from foods;

max(id)
-------
416

sqlite> insert into foods values (416, 1, 'Chocolate Bobka');
SQL error: PRIMARY KEY must be unique
```

更新记录

update 命令用于更新表中的记录，该命令可以修改一个表中一行或多行中的一个或多个字段。update 语句的一般格式为：

```
update table set update_list where predicate;
```

update_list 是一个或多个“字段赋值”的列表，字段赋值的格式为 column_name=value。where 子句的用法与 select 语句相同，实际上，update 语句的一半都是 select 语句。where 子句使用断言识别要修改的行，然后将更新列应用到这些行，例如：

```
update foods set name='CHOCOLATE BOBKA'
where name='Chocolate Bobka';
select * from foods where name like 'CHOCOLATE%';
```

```
id     type_   name
-----  -----   ----------------------------
10     1       CHOCOLATE BOBKA
11     1       Chocolate Eclairs
12     1       Chocolate Cream Pie
222    9       Chocolates, box of
223    9       Chocolate Chip Mint
224    9       Chocolate Covered Cherries
```


update 命令非常简单直接，它有很多扩展。正如 insert 语句一样，必须注意 unique 约束，因为也可能停止更新。

```
sqlite> update foods set id=11 where name='CHOCOLATE BOBKA';
SQL error: PRIMARY KEY must be unique
```

对于任何约束，这种情况都会发生。

删除记录

使用 delete 命令可以删除表中的记录。delete 语句的一般格式为：

```
delete from table where predicate;
```

同样，delete 与 update 有些类似。就如同从 update 语句中移出了 set 部分，然后将 update 替换成 delete。where 子句的用法与 select 语句相同，只是在 delete 语句中，where 是用来确定需要被删除的记录。如：

```
delete from foods where name='CHOCOLATE BOBKA';
```

数据完整性

数据完整性用于定义和保护表内部或表之间的数据关系。一般有四种完整性：域完

整性、实体完整性、引用完整性和用户定义完整性。域完整性涉及控制字段内的值。实体完整性涉及表中的行。引用完整性涉及表之间的行——具体地讲，就是外键关系。用户定义完整性可以包罗万象。

数据完整性是通过约束实现的。约束就是对字段存储值的一种限制措施。数据库会对字段中的存储值进行完整性约束强制实施。SQLite 中，约束还包括对冲突解决的支持，冲突解决也会在本章后面讨论。

暂时先离开 foods 数据库一会，让我们先看看第 3 章引入的 contacts 表。contacts 表的定义如下：

```
create table contacts (
id integer primary key,
name text not null collate nocase,
phone text not null default 'UNKNOWN',
unique (name,phone) );
```

正如您知道的，约束也是表定义的一部分。它们可以和字段定义或者表定义实体关联起来。字段级的约束包括 not null、unique、primary key、foreign key、check 和 collate。表一级的约束包括 primary key、unique 以及 check。所有这些约束都会在本章后面根据所属类型一一讨论。

现在您已经熟悉了 SQLite 的 update、insert 和 delete 命令，许多约束的操作可以派上用场了。正如这些命令对数据的操作一样，约束也对数据进行操作，确保命令修改的数据符合表中定义的约束规则。

实体完整性

关系理论也是包括 SQLite 在内的绝大多数数据库实现的，都要求数据库的所有域都是唯一的、可以定位的。域要可寻址，它相应的行必须是可以寻址的。因此，行必须在某种方式上是唯一的。这就是主键的功能。

主键由至少带有 unique 约束的一个或一组字段组成。unique 约束正如你所见到的要求本字段（或一组字段）的所有值必须是明显可区分的。因此，主键确保每一行在某种方式上与表中的其他行是不同的，最终确保了所有的字段都是可寻址的。实体完整性基本上保持表中的数据是有组织的。如果您找不到数据，那还有什么意义？

唯一性约束

因为唯一性（unique）约束是主键的基础，所以先介绍它。一个 unique 约束要求一个或一组字段的所有值互不相同，或者说唯一。如果试图插入一个重复值，或将一个值更新成一个已存在的值，数据库将引发一个约束非法，并终止操作。唯一性约束可以在字段级或表级定义。在表级定义时，唯一性可以应用到多个字段中。这种情况下，这些字段值的联合必须是唯一的。在 contacts 表中，对 name 和 phone 的联合有唯一性约束。看看试图插入 name 为“Jerry”、电话为“UNKNOWN”的记录时，会发生什么：

```
sqlite> insert into contacts (name,phone) values ('Jerry','UNKNOWN');
SQL error: columns name, phone are not unique

sqlite> insert into contacts (name) values ('Jerry');
SQL error: columns name, phone are not unique

sqlite> insert into contacts (name,phone) values ('Jerry', '555-1212');
```

第一种情况明确指定了 name 和 phone。它匹配已经存在的记录，同时唯一性约束开始生效并终止该操作。第三个 insert 说明唯一性约束应用在 name 和 phone 的联合上，因此，没有报错，因为 name 自身不是唯一的，但是 name 和 phone 合起来是唯一的。

NULL 和 UNIQUE

第 3 章了解了 NULL，本章新学习了 unique 约束。那么定义为 unique 约束的字段可以放入多少个 NULL？理论上看，您可以放入任意多个 NULL。记住：NULL 不等于任何值，甚至不等于其他 NULL。这也是 SQLite 处理定义为 unique 的字段中 NULL 的基础。其他主要的数据库也是这样做的，比如 PostgreSQL 和 Oracle。但是数据库业内在这个问题上有分歧。Informix、Sybase 和微软 SQL Server 只能插入一个 NULL，而 DB2 干脆彻底禁止插入 NULL。

主键约束

在 SQLite 中，当您定义一个表时总要确定一个主键，不管您自己有没有定义。这个字段是一个 64-bit 整型字段，称为 rowid。它还有两个别名——_rowid_和 oid，用这两个别名同样可以取它的值，它的默认取值按照增序自动生成。SQLite 为主键字段提供自动增长特性。

SQLite 为主键提供自增长功能，您可以在定义时使用。如果定义字段类型为 integer

primary key，SQLite 将为该字段创建默认值，该默认值确保整数值是唯一的。实际上，该字段就是 rowid 的别名。它们会引用同样的值，因为 SQLite 使用 64-bit 单符号整数主键。因此，该字段的最大值是 9,223,372,036,854,775,807。

当您到达最大值时，SQLite 会开始搜索该字段还未使用的值，并作为将要插入的值。从表中删除记录时，rowid 可能被回收并在后面的插入中使用。因此，新创建的 rowid 不一定是按照严格顺序增长的。

注意：应该记住我们在第 3 章讨论 order by 时关系型数据没有隐式的顺序。这里再次提醒，不要认为或者假定关系数据库（例如 SQLite）的数据具有某种特定的顺序——即使直觉告诉您这样做是安全的。

如果想要 SQLite 使用唯一的自动主键值，而不是“填补空白”，可以在主键定义 integer primary key 中加入关键字 autoincrement。

到目前为止的例子中，您已经向 contacts 表中插入两条记录，两次都未指定 id 值。正如前面提到的，这是因为 id 定义成 integer primary key。因此，SQLite 为 insert 语句自动生成 id 值。下面可以查询验证：

```
sqlite> select * from contacts;
```



```
id    name    phone
---   -----   --------
1     Jerry   UNKNOWN
2     Jerry   555-1212
```

注意，可以从除了 id 之外的上述别名访问主键：

```
sqlite> select rowid, oid,_rowid_,id, name, phone from contacts;

Id   id   id   id   name    phone
--   --   --   --   ----    -----
1    1    1    1    Jerry   UNKNOWN
2    2    2    2    Jerry   555-1212
```

如果在 integer primary key 后面加上 autoincrement，SQLite 使用不同的主键生成算法，该算法基本上阻止 rowid 回收。它为 insert 只产生新的（不是回收的）rowid。创建表时，如果字段包含 autoincrement 约束，SQLite 会在一个名为 sqlite_sequence 的系统表中记录该字段当前的最大值。它在后面的 insert 语句中只使用比当前值大的，如果达到

了绝对最大值，SQLite 会在后面的 insert 语句中返回 SQLITE_FULL，例如：

```
sqlite> create table maxed_out(id integer primary key autoincrement, x text);
sqlite> insert into maxed_out values (9223372036854775807, 'last one');
sqlite> select * from sqlite_sequence;

name         seq
----------   -------------------
maxed_out    9223372036854775807

sqlite> insert into maxed_out values (null, 'will not work');
SQL error: database is full
```

本例中，我们提供了主键值。SQLite 会将该值作为当前最大值（maxed_out.id）存储在 sqlite_sequence 中。我们提供了 64 位的最大值，在后面的 insert 语句中，如果使用单步增长自动产生值，单步增长的值会达到最大限制（重新回到 0），此时，SQLite 会返回 SQLITE_FULL 错误。

虽然 SQLite 在系统表 sqlite_sequence 中追踪 autoincrement 字段的最大值，但是它不会阻挡您在 insert 语句中自己提供值，只是要求您提供的值必须是该字段唯一的。例如：

```
sqlite> drop table maxed_out;
sqlite> create table maxed_out(id integer primary key autoincrement, x text);
sqlite> insert into maxed_out values(10, 'works');
sqlite> select * from sqlite_sequence;

name         seq
----------   ----------
maxed_out    10

sqlite> insert into maxed_out values(9, 'works');
sqlite> select * from sqlite_sequence;

name         seq
----------   ----------
maxed_out    10

sqlite> insert into maxed_out values (9, 'fails');
SQL error: PRIMARY KEY must be unique

sqlite> insert into maxed_out values (null, 'should be 11');
sqlite> select * from maxed_out;
```

```
id          x
----------  ------------
9           works
10          works
11          should be 11

sqlite> select * from sqlite_sequence;

name        seq
----------  ----------
maxed_out   11
```

这里放弃并重新创建了表 maxed_out，同时显式地插入 rowid 等于 10 的记录。SQLite 允许您插入比 10 小的 rowid，但如果再次试图插入同样的值，因为唯一性约束的特点，SQLite 就会报错。最后，如果您插入了另一个记录并使用系统默认的生成值，SQLite 提供的下一个值是 10+1。

总之，autoincrement 阻止 SQLite 回收主键值（rowid），并在 rowid 达到 64 位最大值时停止，该功能是一些特定的要求。如果应用程序有这样的需求，将自增长字段作为整型主键也许是很好的建议。

像唯一性约束一样，主键约束也可以定义在多个字段中。主键也不一定要使用整型值，如果选择使用其他值，SQLite 内部仍然维护 rowid 字段，它也在您定义的主键上实施唯一性约束，例如：

```
sqlite> create table pkey(x text, y text, primary key(x,y));
sqlite> insert into pkey values ('x','y');
sqlite> insert into pkey values ('x','x');
sqlite> select rowid, x, y from pkey;

rowid       x           y
----------  ----------  ----------
1           x           y
2           x           x

sqlite> insert into pkey values ('x','x');
SQL error: columns x, y are not unique
```

97

本例中的主键从技术上讲横跨两个字段，具有唯一性约束，因为 SQLite 总是在内部维护 rowid。许多数据库设计专家建议使用实际字段作为主键字段，本书作者也建议

在合适的地方这样做。

域完整性

最简单的域完整性定义就是字段的值遵从赋予的规定。也就是说，字段中的值必须是字段定义范围内的。但是，域（domain）这个词有点模糊。很多人常用域与编程语言中的类型做比较，例如字符串、浮点数。虽然这个比较有合理之处，但是实际上域完整性比编程语言的类型范围广得多。

从最简单的类型开始了解域约束，例如整型可以为字段添加额外的约束创建更严格的可接受值范围。例如，可以创建一个整型字段添加约束，该字段只能是三个值{-1,0,1}。这种情况下，就是修改可接受值的范围（域的范围从所有的整型值变为只有三个值），而不是数据类型本身。域处理两件事情：类型和范围。

考虑另一个事例：contacts 表中的 name 字段，其定义如下：

```
name text not null collate nocase
```

其中，text 域定义类型和初始的可接受值范围，遵从定义的 text 范围受到限制。name 字段域是除了 null 值之外的所有 text，并且大小写是不敏感的。它依然是 text，可以像操作文本一样操作它，但是它可接受值的范围与 text 有所不同。

您可能会说，列的域与列的类型不是一回事。列的域是由两个概念组成的：类型和范围。列的类型定义了其值是如何展开和如何操作的——如何存储并对它们进行排序、搜索、加、减，等等。列的范围就是它能存储的可接受值的集合，范围不必与列的声明类型相同。类型的范围代表了值的最大范围。如您所见，列的范围可能通过约束进行限制。所以出于实用目的，你可以将列的域想象成被施加了约束的类型。

类似地，域完整性有两个组成部分：类型检查和范围检查。虽然 SQLite 支持许多标准的域约束来检查范围（not null、check 等），但是类型检查方法也正是 SQLite 与其他数据库不同的地方。实际上，SQLite 的类型和类型检查方法也是它最具争议、容易误解的功能之一。第 11 章在讨论 SQLite 内核时会深入讲解该功能的基础。

现在开始，我们讨论比较容易的内容：默认值、not null 约束、check 约束以及排序规则。

默认值

如果用 insert 语句插入记录时没有为该字段指定值，关键字 default 将为字段提供一个默认值。default 只是一个约束，它保证该字段有值，并在需要时出现。它符合域完整性，因为它为处理字段中的 null 值提供一种策略。如果字段没有默认值，并在 insert 语句中没有提供指定值，SQLite 将为该字段自动赋值 null。例如，contacts 的 name 字段默认值为'UNKNOWN'，考虑如下例子：

```
sqlite> insert into contacts (name) values ('Jerry');
sqlite> select * from contacts;

id          name        phone
----------  ----------  ----------
1           Jerry       UNKNOWN
```

用 insert 命令插入一行，并为 name 字段指定值，但是 phone 字段没有。正如结果所示，phone 字段的默认值生效了，提供了'UNKNOWN'。如果字段 phone 没有默认值，在本例中，该行 phone 字段的值将会是 NULL。

default 还可以接受 3 种预定义格式的 ANSI/ISO 保留字，用于生成日期和时间值。current_time 将会生成 ANSI/ISO 格式（HH:MM:SS）的当前时间。current_date 会生成当前日期（格式为 YYYY-MM-DD）。current_timestamp 会生成一个日期时间的组合（格式为 YYYY-MM-DD HH:MM:SS）。例如：

```
create table times ( id int,
  date not null default current_date,
  time not null default current_time,
  timestamp not null default current_timestamp );

insert into times (id) values (1);
insert into times (id) values (2);
select * from times;
```

```
id    date        time        timestamp
---   ----------  ----------  -------------------
1     2010-06-15  23:30:25    2010-06-15 23:30:25
2     2010-06-15  23:30:40    2010-06-15 23:30:40
```

这些默认值对于需要记录日志或时间戳事件非常有用。

NOT NULL 约束

如果您不喜欢使用 NULL 值，那么 NOT NULL 约束就是专为您而设计的。NOT NULL 约束可以确保该字段值不为 NULL。insert 语句不能向该字段插入 NULL，更新语句也不能将该字段已存在的值修改为 NULL。常常在 insert 语句看见 NOT NULL 产生的报错。具体地讲，没有默认值的 NOT NULL 约束阻止未给该字段赋值的 insert 语句执行（因为系统提供的默认值是 NULL）。前面的例子中，name 字段的 NOT NULL 约束要求 insert 语句必须为该字段提供值。例如：

```
sqlite> insert into contacts (phone) values ('555-1212');
SQL error: contacts.name may not be NULL
```

插入命令为 phone 字段指定值，但是 name 字段没有。name 字段的 NOT NULL 约束将会启动并禁止该操作。

处理未知数据和 NOT NULL 约束的实用方法是给字段设定默认值。例如，phone 字段，虽然该字段有 NOT NULL 约束，但是它也有默认值。如果 insert 语句没有为 phone 字段指定值，默认约束会生效并提供“UNKNOWN”值，因此，也满足 NOT NULL 约束。通常，大家喜欢将 default 约束和 NOT NULL 约束放在一起使用，这样 insert 语句既可以安全使用默认值，也可以将 NULL 排除在外。

check 约束

check 约束允许定义表达式来测试要插入或者更新的字段值。如果该值不满足设定的表达式标准，数据库会报约束违反错误。因此，您可以定义除了 unique 或 NOT NULL 之外的数据完整性以满足特定的应用。例如，check 约束可能确保电话号码字段的值至少是 7 位字符长度。要实现这样的约束，可以在 phone 字段的定义上添加如下约束，或者作为表定义中的一条单独约束：

```
create table contacts
( id integer primary key,
name text not null collate nocase,
phone text not null default 'UNKNOWN',
unique (name,phone),
check (length(phone)>=7) );
```

本例中，任何试图插入或者更新语句中 phone 的值如果小于 7 个字符，都会导致违反约束。在 check 约束中使用表达式可以像 where 子句中一样（子查询例外）。例如，foo

表的定义可以如下：

```
create table foo
( x integer,
y integer check (y>x),
z integer check (z>abs(y)) );
```

在该表中，z 字段的每一个值必须大于对应 y 字段的值，当然也大于对应 x 字段的值，如下所示：

```
insert into foo values (-2, -1, 2);
insert into foo values (-2, -1, 1);
SQL error: constraint failed

update foo set y=-3 where x=-3;
SQL error: constraint failed
```

所有字段的 check 约束都是在修改发生前评估的。要想修改成功，所有约束的表达式必须判断为真。

从功能上看，触发器可以与 check 约束一样有效地实现数据完整性。事实上，触发器的功能更强大。如果发现不能在 check 约束中完全表达您所需要的内容，可以试试用触发器。本章后面的“触发器”一节会有详细讨论。

100

外键约束

SQLite 支持关系理论中的关系完整性概念。与绝大多数数据库一样，它通过外键约束支持这些概念。关系完整性也叫外键，它确保了一个表中的关键值必须从另一个表中引用，且该数据必须在另一个表中实际存在。经典的例子就是父子关系、主从关系，以及订单和商品这样的关系，甚至本书使用的数据库中食品表和食品类型表之间的关系。

SQLite 支持在创建表的时候使用外键，可以使用如下语法（此处为了方便阅读，有所简化）：

```
create table table_name
( column_definition references foreign_table (column_name)
  on {delete|update} integrity_action
  [not] deferrable [initially {deferred|immediate}, ]
… );
```

该语法看起来有些难，但实际上可以划分为三个基本部分。下面以 foods 数据库为

例，foods 表和 food_types 表当前的定义如下：

```
CREATE TABLE food_types(
  id integer primary key,
  name text );

CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
```

可以看出，food_types 表的 id 可以唯一识别（参考前面关于主键的讨论）。foods 表使用了 type_id 字段，引用自表 food_types。如果想要使用引用完整性确保 SQLite 保护您的数据，确保 foods 表的 type_id 值都存在于 food_types 的 id 字段，那么可以使用如下方式定义 foods 表：

```
create table foods(
  id integer primary key,
  type_id integer references food_types(id)
  on delete restrict
  deferrable initially deferred,
  name text );
```

两个定义的不同之处用黑体字显示了出来，使读者很容易理解。外键的第一步告诉 SQLite type_id 引用自表 food_types 的 id 字段。现在，就有完整性约束，它可以完成这些艰苦的工作。本例中，使用了 delete restrict，该命令告诉 SQLite 阻止任何这样的删除行为，如果从表 food_types 中删除某行，将导致 foods 表的 id 没有父 id 的存在。完整的规则定义如下：

set null：如果父值被删除或者不存在了，剩余的子值将改为 null。

set default：如果父值被删除或者不存在了，剩余的子值将修改为默认值。

cascade：更新父值时，更新所有预知匹配的子值。删除父值时，删除所有的子值。要特别注意该选项，cascade 的删除功能可能会出现意想不到的效果。

restrict：更新或删除父值可能会出现孤立的子值，从而阻止（终止）事务。

no action：使用一种松弛的方法，不干涉操作执行，只是观察变化。在整个语句（如果约束定义为 deferred 的事务也一样）的结尾报出错误。

最后，SQLite 支持 deferrable 子句，该子句控制定义的约束是立即强制实施还是延

迟到整个事务结束时。

排序规则

排序规则涉及文本值如何比较。不同的排序规则使用不同的比较方法。例如，有些排序规则可能是不区分大小写的，因此，“JujyFruit”和“JUJYFRUIT”是一样的。另一些排序规则可能是大小写敏感的，这种排序规则中，“JujyFruit”和“JUJYFRUIT”将是不同的。

SQLite 有三种内置的排序规则，默认的是二进制排序规则，该规则使用 C 函数 memcmp()逐字节比较文本值。对于许多西方语言，例如英语，这样是合适的。第二种，nocase 是拉丁字母中使用的 26 个 ASCII 字符的非大小写敏感排序算法。第三种，是 reverse 排序规则，它与二进制排序规则相反。reverse 被更多地用来测试（也有用来说明的）。

SQLite 的 C API 提供定制自定义排序规则的方法。该功能允许开发者支持二进制排序规则不能很好适用的语言或者语系。具体内容参看第 7 章。

关键字 collate 定义字段的排序规则。例如 contacts.name 中定义的排序是 nocase，意思是它是大小写不敏感的。因此，如果试图插入 name 为“JERRY”、phone 为“555-1212”的内容，将会失败：

```
sqlite> insert into contacts (name,phone) values ('JERRY','555-1212');
SQL error: columns name, phone are not unique
```

根据 name 字段的排序规则，“JERRY”与“Jerry”是相同的，表中已经存在该值。因此，新的 name 等于“JERRY”的行，将被认为是重复值。SQLite 排序规则默认是大小写敏感的。如果 name 字段没有定义 nocase，那么前面的语句就可以正常插入。

存储类

如前文所述，SQLite 在处理数据类型时与其他的数据库不同，区别在于它所支持的类型以及这些类型是如何存储、比较、强化（enforced）和指派（assign）的。下面将介绍 SQLite 存储类的基础。第 11 章会介绍 SQLite 的内核，介绍本质上不同但又灵活的数据类型方法。 102

一般情况下，SQLite 有 5 个原始的数据类型，被称为存储类。存储类通常是指值在磁盘上存储的格式，其实就是类型或数据类型的同义词。表 4-1 列出了这 5 个存储类及

其说明。

表 4-1　SQLite 存储类

名　称	说　明
integer	整数值是全部的数字（包括正和负）。整数可以是 1、2、3、4、6 或 8 字节。整数的最大范围（8 字节）是{-9223372036854775808, -1,0,1 +9223372036854775807}。SQLite 根据数字的值自动控制整数所占的字节数
real	实数是十进制的数值。SQLite 使用 8 字节的浮点数来存储实数
text	文本（Text）是字符数据。SQLite 支持几种字符编码，包括 UTF-8 和 UTF-16（大字节头和小字节头）。字符串的最大值在编译和运行时可以调整，默认最大值是 1,000,000,000 字节
blob	二进制大对象（BLOB）数据可以是任意类型的数据。BLOB 的最大值在编译和运行时可以调整，默认最大值是 1,000,000,000 字节
NULL	NULL 表示没有值。SQLite 具有对 NULL 的完全支持

SQLite 通过值的表示法来判断其类型，下面就是 SQLite 使用的推断方法：

- SQL 语句中用单引号或双引号可起来的文字被指派为 text。
- 如果文字是未用引号引起来的数据，并且没有小数点和指数，被指派为 integer。
- 如果文字是未用引号引起来的数据，并且带有小数点或指数，被指派为 real。
- 用 NULL 说明的值被指派为 NULL 存储类。
- 如果一个值的格式为 x'ABCD'，其中 ABCD 为 16 进制数字，则该值被指派为 blob。其中，x 前缀大小写皆可。

SQL 函数 typeof()根据值的表示法返回其存储类。使用该函数后，下面 SQL 语句返回的结果为：

103
```
sqlite> select typeof(3.14), typeof('3.14'),
        typeof(314), typeof(x'3142'), typeof(NULL);

typeof(3.14)  typeof('3.14')   typeof(314)  typeof(x'3142')  typeof(NULL)
------------  ---------------  -----------  ---------------  ------------
real          text             integer      blob             null
```

这是 5 种内部的存储类，其中的 3.14 看起来就像实数，因此它就是实数。'3.14'看起来就像文本，因此就是文本，以此类推。

SQLite 中单独的一个字段可能包含不同存储类的值。这也是 SQLite 处理数据与其他数据库不同的第一个地方，许多熟悉其他数据库的人常常会问到“什么，一个字段可

以有不同的存储类型？”请看下面的示例：

```
sqlite> drop table domain;
sqlite> create table domain(x);
sqlite> insert into domain values (3.142);
sqlite> insert into domain values ('3.142');
sqlite> insert into domain values (3142);
sqlite> insert into domain values (x'3142');
sqlite> insert into domain values (null);
sqlite> select rowid, x, typeof(x) from domain;

rowid       x           typeof(x)
----------  ----------  ----------
1           3.142       real
2           3.142       text
3           3142        integer
4           1B          blob
5           NULL        null
```

这种操作会带来一些问题。这种字段中的值如何存储和比较？如何对一个包含了 integer、real、text、blob 和 null 值的字段排序？一个整数和一个 blob 如何比较？哪个更大？它们能相等吗？

答案是：具有不同存储类的值可以存储在同一个字段中。可以被排序，因为这些值可以相互比较。SQLite 实现了完善的定义规则来做这件事。不同存储类的值可以通过它们各自类的“类值”进行排序，定义如下：

1. NULL 存储类具有最低的类值。一个具有 NULL 存储类的值比所有其他的值都小（包括其他具有 NULL 存储类的值）。在 NULL 值之间，没有具体的排序顺序。
2. integer 或 real 存储类值高于 NULL，它们的类值相等。integer 值和 real 值通过其数值进行比较。
3. text 存储类的值比 integer 和 real 高。数值永远比字符串的值低。当两个 text 值进行比较时，其值大小由该值中定义的“排序法”决定。
4. blob 存储类具有最高的类值。具有 blob 类的值大于其他所有类的值。blob 值之间在比较时使用 C 函数 memcmp()。

所以，当 SQLite 存储一个字段时，它首先按存储类分组——NULL 第一，然后是 104

integer 和 real，接着是 text，最后是 blob。然后进行分组内的排序，NULL 类内部各值不必排序，integer 和 real 按照数字比较，text 按照恰当的排序算法比较，blob 使用 memcmp()——实际上是比较比特位。图 4-1 说明了一个假设字段如何按照升序排序。

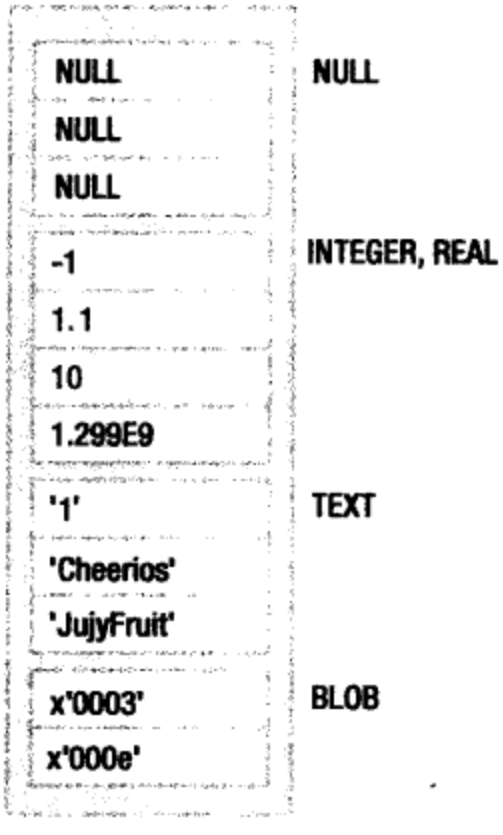


图 4-1　存储类排序顺序

如果您有机会实际使用 SQLite 存储类，可能需要再重读本节，以加深对 SQLite 这方面的理解。在第 11 章中，当学习完存储类的内核、显化类型、近似类型以及与类型和存储类相关的引擎内核后，可能对本节理解更深刻。

视图

视图即虚拟表，也称为派生表，因为它们的内容都派生自其他表的查询结果。虽然视图看起来感觉就像基本表一样，但是它们不是基本表。基本表的内容是持久的，而视图的内容是在使用时动态产生的。创建视图的语法如下：

```
create view name as select-stmt;
```

视图的名称通过 name 指定，其定义由 select-stmt 定义。最终生成的视图看起来就像名为 name 的基本表。想象一下您有一个一直在运行的查询，您会对多次编写这样的

查询感到厌烦，此时视图则是这种情况的最佳选择。例如：


```
select f.name, ft.name, e.name
from foods f
inner join food_types ft on f.type_id=ft.id
inner join foods_episodes fe on f.id=fe.food_id
inner join episodes e on fe.episode_id=e.id;
```

该查询返回所有 food 的名称，以及所在系列。它是一个多达 504 行的大表，但不用每次需要这些结果时写出前面的查询，我们可以将它们整理成命名为 details 的视图形式。

```
create view details as
select f.name as fd, ft.name as tp, e.name as ep, e.season as ssn
from foods f
inner join food_types ft on f.type_id=ft.id
inner join foods_episodes fe on f.id=fe.food_id
inner join episodes e on fe.episode_id=e.id;
```

现在您可以像查询表一样查询 details，例如：

```
sqlite> select fd as Food, ep as Episode
        from details where ssn=7 and tp like 'Drinks';

Food                Episode
------------------- -------------------
Apple Cider         The Bottle Deposit 1
Bosco               The Secret Code
Cafe Latte          The Postponement
Cafe Latte          The Maestro
Champagne Coolies   The Wig Master
Cider               The Bottle Deposit 2
Hershey's           The Secret Code
Hot Coffee          The Maestro
Latte               The Maestro
Mellow Yellow soda  The Bottle Deposit 1
Merlot              The Rye
Orange Juice        The Wink
Tea                 The Hot Tub
Wild Turkey         The Hot Tub
```

视图的内容是动态生成的。因此，每次使用 details 时，基于数据库的当前数据执行

相关的 SQL 语句，产生结果。其他数据库通常不提供视图的其他功能，例如基于视图的安全，但是 SQLite 提供，使用操作控制工具与 SQLite 编程，就可提供基于视图的安全。相关内容将在第 5、6 章讨论。

最后，使用命令 dorp view 删除视图。

```
drop view name;
```

通过提供 name 删除对应的视图。

可更新的视图

关系模型需要可更新的视图。您可以修改这些视图，可以对这些视图进行 insert 或 update 操作，例如，这些变化可以直接应用到底层表中。SQLite 不支持可更新的视图。但是，使用触发器可以创建类似的东西。后面的“触发器”一节将展示该技术。

索引

索引是一种用来在某种条件下加速查询的结构。考虑如下查询：

```
SELECT * FROM foods WHERE name='JujyFruit';
```

当数据库搜索匹配的行时，执行这种查询的默认方法是调用顺序扫描。也就是说，逐个搜索（扫描）表中的所有行，查看 name 属性是否匹配“JujyFruit”。

但是，如果频繁使用这种查询，并且表 foods 非常大，使用索引的方法查找数据就更有意义了。与其他许多数据库类似，SQLite 使用 B-tree 做索引。

索引也会增加数据库的大小。从字面上理解，它们复制了一份所索引的字段。如果多表中的所有字段都创建索引，表的大小可能翻倍。另一个要考虑的情况是索引的维护。进行 Insert、update 和 delete 操作时，除了修改表，数据库也必须修改对应的索引。虽然索引可以加速查询，但是它们可能降低 insert、update 和类似操作的速度。

创建索引的命令如下：

```
create index [unique] index_name on table_name (columns)
```

变量 index_name 是索引的名称，table_name 是包括索引所在字段的表名称，变量 columns 是一个字段或以逗号分隔的多个字段。

如果使用关键字 unique，将会在索引上添加约束，索引中的所有值必须是唯一的。

这不仅适用于索引，也适用于索引所在字段。unique 关键字覆盖 index 中的所有字段，不管是联合值还是单个值，都必须是唯一的。例如：

```
sqlite> create table foo(a text, b text);
sqlite> create unique index foo_idx on foo(a,b);
sqlite> insert into foo values ('unique', 'value');
sqlite> insert into foo values ('unique', 'value2');
sqlite> insert into foo values ('unique', 'value');
SQL error: columns a, b are not unique
```


可以看到，唯一性约束不仅存在于单个字段，也存在于联合字段。注意，排序规则在这发挥了重要作用。

若要删除索引，可使用 drop index 命令，该命令定义如下：

```
drop index  index_name;
```

排序规则

索引中的每个字段都有相关的排序规则，例如，要在 foods.name 上创建大小写不敏感的索引，可以使用如下命令：

```
create index foods_name_idx on foods (name collate nocase);
```

这意味着 name 字段的值将按照大小写无关进行排序。可以在 shell 中使用 SQL 命令.indices 列出表中的索引。例如：

```
sqlite> .indices foods
foods_name_idx
```

也可以用.schema 命令查看更多的信息。

```
sqlite> .schema foods
CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
```

还可以通过查询表 sqlite_master 获取相关信息，这将在后面讨论。

使用索引

了解何时使用索引以及何时不用索引是非常重要的。SQLite 中有一些具体的条件来

判断是否使用索引。如果可以的话，对于下面会在 where 子句中出现的表达式，SQLite 将使用单个字段索引。

```
column {=|>|>=|<=|<} expression
expression {=|>|>=|<=|<} column
column IN (expression-list)
column IN (subquery)
```

在使用之前，多字段索引有更复杂的条件。下面通过例子可以更好地说明，假设有如下表定义：

```
create table foo (a,b,c,d);
```

108

此外，创建如下多字段索引：

```
create index foo_idx on foo (a,b,c,d);
```

foo_idx 字段的顺序是从左到右的。也就是说，在如下查询中：

```
select * from foo where a=1 and b=2 and d=3
```

只有第一个和第二个条件将使用索引。不使用第三个条件的原因是没有条件使用 c 去缩小到 d 的差距。基本上，当 SQLite 使用多字段索引时，它从左到右智能地使用字段，即从左边的字段开始，查询使用字段的条件，然后移动到第二个字段，以此类推，一直继续，直到 where 子句中无法找出有效条件。

```
select * from foo where a>1 and b=2 and c=3 and d=4
```

SQLite 会在字段 a 上执行索引扫描，表达式 a>1 称为最右边的索引字段，因为它使用了不等于号。之后的所有字段作为结果是不合格的。类似地，如下语句使用索引字段 a 和 b，在 b>2 处停止，因为它是最右的不等号索引字段。

```
select * from foo where a=1 and b>2 and c=3 and d=4
```

最后，创建索引时，一定要有理由确保可以获得性能的改善。选择好的索引是非常重要的；分布凌乱的索引可能会导致希望获得良好的性能愿望落空。

触发器

当具体的表发生特定的数据库事件时，触发器执行对应的 SQL 命令。创建触发器

的一般命令如下：

```
create [temp|temporary] trigger name
[before|after] [insert|delete|update|update of columns] on table
action
```

触发器是通过名称、行为和表定义的。行为（或者称为触发体）由一系列 SQL 命令组成，当某些事件发生时，触发器负责启动这些命令。此外，可以通过关键字 before 或 after 来指定这些操作是在事件发生前还是发生后执行。事件包括在具体的表中执行的 delete、insert 和 update 命令。触发器可以用来创建自定义完整性约束、日志改变、更新表和其他很多事情。触发器的作用只限于所写的 SQL 命令。

更新触发器

与 insert 和 delete 触发器不同，update 触发器可以在表的执行字段上定义，这种触发器的一般格式如下：

```
create trigger name
[before|after] update of column on table
action
```

下面是一个 update 触发器的 SQL 脚本。

```
create temp table log(x);

create temp trigger foods_update_log update of name on foods
begin
  insert into log values('updated foods: new name=' || new.name);
end;

begin;
update foods set name='JUJYFRUIT' where name='JujyFruit';
select * from log;
rollback;
```

该脚本创建一个名为 log 的临时表，以及在表 foods 的 name 字段上创建一个临时的 update 触发器，该触发器执行时会向 log 表插入一条消息。该行为发生在之后的事务内部，事务的第一步更新 name 为“JUJYFRUIT”所在行的 name 字段，这将促使 update 触发器执行。当触发器执行时，它向 log 插入一条记录。事务的下一步读取 log，显示触发器确实执行了，然后事务回滚刚才的改变，当连接结束时，销毁 log 表和 update 触

发器。运行脚本产生如下输出：

```
# sqlite3 foods.db < trigger.sql
```

```
create temp table log(x);

create temp trigger foods_update_log after update of name on foods
begin
  insert into log values('updated foods: new name=' || new.name);
end;

begin;
update foods set name='JUJYFRUIT' where name='JujyFruit';
SELECT * FROM LOG;
x
-------------------------------------------------
updated foods: new name=JUJYFRUIT
rollback;
```


SQLite 提供在 update 触发器中对未更新行（最初的行）和已更新行的访问。未更新行可以引用为 old，已更新行可引用为 new。注意脚本中的触发器是如何引用 new.name 的。old 和 new 行的所有属性都可以通过点号来引用，例如 new.type_id 或 old.id。

错误处理

定义为事件发生前执行的触发器有机会阻止事件的发生，同样，定义为事件发生后执行的触发器可以检查事件，具备重新思考的机会。before 和 after 触发器可以使您具备实施新的完整性约束。SQLite 提供一个特殊的 SQL 函数 raise()供触发器调用，该函数允许在触发器内产生错误。raise()定义如下：

```
raise(resolution, error_message);
```

第一个参数是冲突解决策略（abort、fail、ignore、rollback 等）。第二个参数是错误消息。如果使用 ignore，当前触发器的剩余语句、促使触发器执行的 SQL 命令，以及将被触发执行的触发器，都将终止。如果促使触发器执行的 SQL 语句本身是另一个触发器的组成者，那么该触发器在触发行为的下一个 SQL 命令处继续执行。

可更新的视图

正如本章前面提到的，触发器使得创建像可更新的视图一类事物成为可能。这里创

建一个视图，然后创建负责处理视图上更新事件的触发器。SQLite 支持视图触发器，可以使用 instead of 替代触发器的定义。下面创建一个连接 foods 和 food_types 的视图：

```
create view foods_view as
  select f.id fid, f.name fname, t.id tid, t.name tname
  from foods f, food_types t;
```

该视图通过外键关系连接两个表。注意，这里为视图中的所有字段创建了别名。这样就可以在触发器内部应用这些字段时区分各自的 id 和 name。现在，通过创建 update 触发器，实现可更新的视图：

```
create trigger on_update_foods_view
instead of update on foods_view
for each row
begin
  update foods set name=new.fname where id=new.fid;
  update food_types set name=new.tname where id=new.tid;
end;
```

现在，如果您试图更新 foods_view，该触发器将被调用。触发器只是简单地获取 update 语句提供的值，使用这些值更新底层的基本表 foods 和 food_types。测试一下：

111

```
.echo on
-- Update the view within a transaction
begin;
update foods_view set fname='Whataburger', tname='Fast Food' where fid=413;
-- Now view the underlying rows in the base tables:
select * from foods f, food_types t where f.type_id=t.id and f.id=413;
-- Roll it back
rollback;
-- Now look at the original record:
select * from foods f, food_types t where f.type_id=t.id and f.id=413;
```

```
begin;
update foods_view set fname='Whataburger', tname='Fast Food' where fid=413;
select * from foods f, food_types t where f.type_id=t.id and f.id=413;
id   type_id   name             id   name
---  -------   --------------   ---  ---------
413  1         Whataburger      1    Fast Food
```

```
rollback;

select * from foods f, food_types t where f.type_id=t.id and f.id=413;
id   type_id   name             id   name
---  -------   ---------------  ---  ---------
413  1         Cinnamon Bobka   1    Bakery
```

现在可以通过添加 insert 和 delete 触发器完成基于触发器通过视图控制数据。

事务

事务定义了一组 SQL 命令的边界，这组命令或者作为一个整体被全部执行，或者都不执行，这被称为数据库完整性的原子性原则。这种关系的典型例子就是银行转账，假设银行程序从一个账户向另一个账户转账，转账程序通过如下方式进行：首先将第一个账户的钱转入第二个账户，然后从第一个账户删除对应的数目；或者首先从第一个账户删除要转账的数目，然后向第二个账户插入对应的数目。无论哪种方式，都是通过两步完成的：先插入，后删除；或者先删除，后插入。

但是在转账期间，如果数据库服务器突然崩溃或者电力中断，第二个操作没有完成怎么办？要么这笔钱存在于两个账户（第一种方式），要么这笔钱在两个账户中都不存在（第二种方式）。无论发生哪种情况，总有人无法接受。数据库也处于不一致的状态，关键是这两步操作必须要同时被执行或者一步都不执行。这就是事务的本质。

事务的范围

事务由 3 个命令控制：begin、commit 和 rollback。begin 开始一个事务，begin 之后
112 的所有操作都可以取消，如果连接终止前没有发出 commit，也会被取消。commit 提交事务开始后所执行的所有操作。类似地，rollback 还原 begin 之后的所有操作。例如：

```
sqlite> begin;
sqlite> delete from foods;
sqlite> rollback;
sqlite> select count(*) from foods;

count(*)
- - - - - - - - -
```

412

上面开始了一个事务，先删除了 foods 表的所有行，但是又用 rollback 进行了回滚。在执行 select 时发现表中没发生任何改变。

默认情况下，SQLite 中每条 SQL 语句自成事务（自动提交模式）。也就是说，如果您没有使用 begin…commit/rollback 定义事务的范围，SQLite 默认每条单独的 SQL 命令就是有 begin…commit/rollback 的事务。这种情况下，所有成功完成的命令都自动提交。同样，所有遇见错误的命令都回滚。这种操作模式（隐式事务）也称为自动提交模式：SQLite 以自动提交模式运行单个命令，如果命令没有失败，那它将自动提交。

SQLite 也支持 savepoint 和 release 命令，这些命令扩展了事务的灵活性，包含多个语句的工作体可以设置 savepoint，回滚可以返回到某个 savepoint。创建 savepoint 和启动 savepoint 命令一样简单，如下所示：

```
savepoint  justincase;
```

如果意识到需要返回到某个地方，不用回滚整个事务，可以使用如下命名回滚：

```
rollback [transaction] to justincase;
```

本例使用 justincase 作为 savepoint 名称，您也可以使用其他名称。

冲突解决

如前所述，违反约束会导致事务的终止。在对数据进行很多修改的过程中，命令终止会造成什么后果？大多数数据库（管理系统）都是简单地将前面所做的修改全部取消。这也是数据库处理违反约束的方式。

SQLite 有其独特的方法允许您指定不同的方式来处理约束违反（或者说从约束违反中恢复），这种功能被称为冲突解决，如下例所示：

```
UPDATE:

sqlite> update foods set id=800-id;
SQL error: PRIMARY KEY must be unique
```

当 update 语句执行到第 388 个记录时，它试图将 id 更新为 800–388=412，但是 id 为 412 的记录已经存在，违反了唯一性约束。因此，该命令终止。但是在违反约束前，SQLite 已经更新了 387 个记录。应该如何处理？默认行为是终止命令并回滚所有的修改， 113

保存事务的完整性。

如果您想让已经修改的 387 个记录继续保留，该怎么办？其实只需要使用恰当的冲突解决方案就行。SQLite 提供 5 种可能的冲突解决方案或策略，它们可以用来解决冲突（约束违反）：replace、ignore、fail、abort 和 rollback。这 5 种方法定义了错误容忍范围或敏感度：从最宽松的 replace，到最严格的 rollback。这些解决方法的定义如下（以严重度排序）：

- **replace**：当违反了唯一性约束时，SQLite 将造成这种违反的记录删除，以插入或修改的新记录替代，SQL 继续执行，且不报错。如果违法了 NOT NULL 约束，使用该字段的默认值代替 NULL。如果该字段没有默认值，SQLite 应用 abort 策略。有一点要特别注意，当冲突解决策略为了满足约束而删除记录时，该行的删除触发器不会被触发。这种行为可能在将来的版本中改变。
- **ignore**：当约束违反发生时，SQLite 允许命令继续执行，违反约束的行保持不变。而它之前和之后的记录都继续修改。也就是说，所有会触发约束违反的行都不动，保持原貌，命令继续处理其他的，且不报错。
- **fail**：当约束违反发生时，SQLite 终止命令，但是不恢复约束违反之前已经修改的记录。也就是说，在约束违法发生前的改变都保留。例如，如果 update 命令在第 100 行违反约束，那么前 99 行已经修改的记录不会回滚。但是对第 100 行和之外的改变不会发生，因为命令已经终止了。
- **abort**：当约束违反发生时，SQLite 恢复命令所做的所有改变并终止命令。abort 是 SQLite 中所有操作的默认解决方法，也是 SQL 标准定义的行为。注意：abort 也是最昂贵的冲突解决策略——要求额外的工作，设置要求没有冲突发生。
- **rollback**：当约束违反发生时，SQLite 执行回滚——终止当前命令和整个事务。最终结果就是当前命令所做的改变和事务中之前命令的改变都被回滚。这也是最严格的冲突解决方法，单个约束违反导致事务中执行的所有操作都回滚。

冲突解决方法既可以在 SQL 命令中指定，也可以在表和索引的定义中执行。具体地讲，冲突解决方法可以在 insert、update、create table 和 create index 中指定。此外，它在触发器何中也有具体含义。冲突解决方法在 insert 和 update 中的语法形式如下：

```
114> insert or resolution into table (column_list) values (value_list);
     update or resolution table set (value_list) where predicate;
```

冲突解决策略紧跟在 insert 或 update 命令后面，并加上前缀 or。insert 或 replace 表达式可以缩写成 replace。这与其他数据库中的“merge”或“upsert”行为类似。

前面的 update 例子中，已经更新的 387 行要回滚，因为使用了默认的 abort 冲突解决方法。如果要继续保留已更新的行，可以使用 fail 冲突解决方法。下面的例子说明了如何使用。该例从表 foods 中复制记录到新表 test 中，并用表 test 做实验。可以在表 test 中添加一个名为 modified 的字段，默认值是 no。在 update 语句中，将其更新为“yes”以追踪那些记录在约束违反发生前更新了。使用 fail 冲突解决方法时，这些更新将保留，可以追踪之后有多少记录被更新了。

```
create unique index test_idx on test(id);
alter table test add column modified text not null default 'no';
select count(*) from test where modified='no';
count(*)
- - - - - - - - - - - - - - - -
412
update or fail test set id=800-id, modified='yes';
SQL error: column id is not unique
select count(*) from test where modified='yes';
count(*)
- - - - - - - - - - - - - - - -
387
drop table test;
```

> ■ 注意：fail 需要额外考虑。记录更新的顺序是不确定的，也就是说，您无法确定记录在表中的顺序或者 SQLite 以何种顺序处理它们。您可能假设它遵从 rowid 字段顺序，但是事实并不总是这样，文档中从未这样说过。再次重申，与任何数据库工作时，不要假设有某种隐式的顺序。如果您正在使用 fail，很多情况下，使用 ignore 会更好。ignore 会完成工作，修改所有可以修改的记录，而不是从第一个约束违反处跳出。

在表内定义时，可以为单个字段指定冲突解决方法，例如：

```
sqlite> create temp table cast(name text unique on conflict rollback);
sqlite> insert into cast values ('Jerry');
sqlite> insert into cast values ('Elaine');
sqlite> insert into cast values ('Kramer');
```

115

cast 表只有一个字段 name，该字段具有唯一性约束，冲突解决方法设置为 rollback。

任何触发该字段约束的 insert 或 update 语句都由 rollback 冲突解决来裁决，而不是默认的 abort。结果不仅是该语句被回滚，而且该语句所在的整个事务都将回滚。

```
sqlite> begin;
sqlite> insert into cast values('Jerry');
SQL error: uniqueness constraint failed

sqlite> commit;
SQL error: cannot commit - no transaction is active
```

本例中的 commit 失败是因为 name 字段的冲突解决方法已经终止了事务。create index 的工作方式类似。当某些字段是约束违反的源头时，表或索引内的冲突解决方法将默认的 abort 改变为这些字段上定义的冲突解决方法。

冲突解决方法是语句级的（DML），可覆盖对象级（DDL）定义的。以前面为例：

```
sqlite> begin;
sqlite> insert or replace into cast values('Jerry');
sqlite> commit;
```

replace 冲突解决方法可以覆盖 cast.name 中定义的 rollback 冲突解决方法。

数据库锁

在 SQLite 中，锁和事务是紧密联系的。要有效地使用事务，需要了解一些关于如何加锁的知识。

SQLite 采用粗粒度的锁。当一个连接要写数据库时，所有其他的连接被锁住，直到写连接结束它的事务。SQLite 有一个加锁表，用来帮助不同的写数据库都能够在最后一刻再加锁，以保证最大的并发性。

预先写日志——SQLite 的未来

即将到来的 SQLite 3.7.0 版本使用预先写日志（Write-Ahead log，WAL）。这将改变事务行为，释放锁，使得 SQLite 不仅只有“一个数据库写”。第 11 章将讨论这些 SQLite 内核即将进行的变化。

116

SQLite 使用锁逐步提升机制，为了写数据库，连接需要逐级获得排它锁。SQLite 有 5 种不同的锁状态：未加锁（unlocked）、共享（shared）、预留（reserved）、未决（pending）

和排它（exclusive）。每个数据库连接在同一时刻只能处于其中一个状态，每种状态（未加锁状态除外）都有一种锁与之对应。

最初的状态是未加锁状态，在此状态下，连接还没有访问数据库。当连接一个数据库，甚至已经用 BEGIN 开始了一个事务时，连接都还处于未加锁状态。

未加锁状态的下一个状态是共享状态。为了能够从数据库中读（不是写）数据，连接必须首先进入共享状态，也就是说，首先要获得一个共享锁。多个连接可以同时获得并保持共享锁，也就是说，多个连接可以同时从同一个数据库中读数据。但哪怕只有一个共享锁还没有释放，也不允许任何连接写数据库。

如果一个连接想要写数据库，它必须首先获得一个预留锁。一个数据库同时只能有一个预留锁，该预留锁可以与共享锁共存，它是写数据库的第 1 阶段。预留锁既不阻止其他拥有共享锁的连接继续读数据库，也不阻止其他连接获得新的共享锁。

一旦一个连接获得了预留锁，它就可以开始处理数据库修改操作了，尽管这些修改只能在缓冲区中进行，而不是实际写到磁盘，对读出内容所做的修改保存在内存缓冲区中（可参考本章后面的“数据库配置”一节中关于 cache_size 编译指示的内容）。

当连接想要提交修改（或事务）时，需要将预留锁提升为排它锁。为了得到排它锁，还必须首先将预留锁提升为未决锁。获得未决锁之后，其他连接就不能再获得新的共享锁了，但已经拥有共享锁的连接仍然可以继续正常读数据库。此时，拥有未决锁的连接等待其他拥有共享锁的连接完成工作并释放其共享锁。

一旦所有的其他共享锁都被释放，拥有未决锁的连接就可以将其锁提升至排它锁，此时就可以自由地对数据库进行修改。所有以前所缓存的修改都会被写到数据库文件中。

死锁

虽然您可能觉得前面关于锁的讨论很有趣，但是您可能也在考虑为什么锁如此重要？为什么需要了解锁的机制呢？如果不了解您正在做什么，您可能会陷入死锁。

考虑表 4-2 中的情况，两个连接 A 和 B 完全不知道对方在同一时刻对同一数据库进行操作。连接 A 启动第一个命令，B 启动第二、第三个命令，A 启动第四个，如此等等。

117

表 4-2　假设的死锁情况

执行顺序	连接 A	连接 B
1	sqlite> begin;	
2		sqlite> begin;
3		sqlite> insert into foo values ('x');
4	sqlite> select * from foo;	
5		sqlite> commit;
6		SQL error: database is locked
7	sqlite> insert into foo values ('x');	
8	SQL error: database is locked	

两个连接都在死锁中结束。B 首先尝试写数据库，也就拥有了一个未决锁。A 再试图写，但当其 insert 语句试图将共享锁提升为预留锁时失败。

为了方便讨论，在此假设连接 A 和 B 都一直在等待数据库可写。那么此时，其他的连接都被锁在外面。如果试图打开第三个连接，它甚至不能读数据库。因为 B 拥有未决锁（它能阻止其他连接获得共享锁）。那么此时，不仅 A 和 B 死锁了，而且它们也将其他人锁在数据库外。基本上，有共享锁和未决锁的那些连接如果不想放弃控制，其他所有的进程都不能再操作此数据库了。

如何避免死锁？当然不能让 A 和 B 坐在会议室通过彼此的律师谈判解决，因为它们甚至不知道彼此的存在。答案是采用正确的事务类型来完成工作。

事务的类型

SQLite 有三种不同的事务类型，它们以不同的锁状态启动事务。事务可以开始于：deferred、immediate 或 exclusive。事务类型在 begin 命令中指定：

```
begin [ deferred | immediate | exclusive ] transaction;
```

一个 deferred 直到必须使用时才获取锁。因此，对于延迟事务，begin 语句本身不会做什么事情——它从未锁定状态开始。这是默认的情况。如果仅仅用 begin 开始一个事务，那么事务就是延迟的，停留在未锁定状态。多个连接可以在同一时刻未创建任何锁的情况下开始延迟事务。这种情况下，第一个对数据库的读操作获取共享锁，类似地，第一个对数据库的写操作试图获取预留锁。

118　由 begin 开始的 immediate 事务在 begin 执行时试图获取预留锁。如果成功，begin

immediate 保证没有其他的连接可以写数据库。正如您知道的，其他的连接可以继续对数据库进行读操作，但是，预留锁会阻止其他新的读取数据库。预留锁的另一个结果是没有其他连接能成功启动 begin immediate 或者 begin exclusive 命令，当其他连接执行上述命令时，SQLite 会返回 SQLITE_BUSY 错误。这时您可以对数据库进行修改操作，但是您还不能提交，当调用 commit 时，会返回 SQLITE_BUSY 错误。这意味着还有其他的读事务没有完成，需要等它们执行完后才能提交事务。

exclusive 事务会试着获取对数据库的排它锁。这与 immediate 工作方式类似，但是一旦成功，exclusive 事务保证数据库中没有其他的活动连接，所以就可对数据库进行任意的读写操作。

前面例子的问题在于两个连接最终都想写数据库，但是它们都没有放弃各自原来的锁，最终，共享锁导致了问题。如果两个连接都以 begin immediate 开始事务，那么死锁就不会发生。在这种情况下，在同一时刻只能有一个连接进入 begin immediate，其他的连接就得等待。必须等待的连接将会不断尝试以确保它最终能开始 immediate 事务。如果所有想对数据库进行写操作的连接使用 begin immediate 和 begin exclusive，那它就提供了一种同步机制，通过这种机制防止了死锁的产生。要使这种方式可以工作，所有的人都必须遵守规则。

基本的准则是：如果使用的数据库没有其他连接，用 begin 就足够了。但是，如果使用的数据库有其他也会对数据库进行写操作的连接，就得使用 begin immediate 或 begin exclusive 开始事务。第 5 章将会讨论事务和锁。

数据库管理

数据库管理一般用于控制数据库如何操作。要执行的许多数据库管理任务都可以通过 SQL 命令完成。SQLite 包括自己一些独特的管理功能，如将多个数据库附加“attach”到一个连接上，以及用来设置各种配置参数的数据库编译指示（pragma）。

附加数据库

SQLite 允许您用 attach 命令将多个数据库“附加”到当前连接上。当您附加了一个数据库时，它的所有内容在当前数据库文件的全局范围内都是可存取的。attach 的语法

如下：

```
attach [database] filename as database_name;
```

其中，filename 指 SQLite 数据库的文件名称和路径，database_name 指要引用的数据库和对象的逻辑名称。主数据库自动赋名为 main。如果创建任何临时的对象，SQLite 会创建一个附加的数据库，并命名为 temp（可以使用后面描述的编辑指示 database_list 查看这些对象）。逻辑名称可以用来引用附加数据库内的对象，与此同时，引用附加数据库内的对象必须提供逻辑名。例如，如果两个数据库都有名为 foo 的表，附加数据库
119 的逻辑名为 db2，查询 db2 中 foo 表的唯一方式是使用全限定名 db2.foo，例如：

```
sqlite> attach database '/tmp/db' as db2;
sqlite> select * from db2.foo;

x
----------
Bar
```

也可以使用 main 全名引用主数据库中的对象：

```
sqlite> select * from main.foods limit 2;

id          type_id     name
----------  ---------   --------------
1           1           Bagels
2           1           Bagels, raisin
```

临时表也可以这样引用：

```
sqlite> create temp table foo as select * from food_types limit 3;
sqlite> select * from temp.foo;

Id    name
---   -------------
1     Bakery
2     Cereal
3     Chicken/Fowl
```

可以使用 detach database 命令将数据库分离，定义如下：

```
detach [database] database_name;
```

该命令使用附加数据库（就是 database_name)）的逻辑名作为参数，分离相关的数据库文件。可以使用 database_list 编译指示获取附加的数据库列表，相关内容在“数据库配置”一节有解释。

数据库清理

SQLite 有两个命令用于清理数据库——reindex 和 vacuum。reindex 用于重建索引，它有如下两种形式：

```
reindex collation_name;
reindex table_name|index_name;
```

第一种形式重建所有使用指定排序名称（collation_name）的索引。当要改变用户定义的排序行为（例如中文中的多排序顺序）时才需要这种形式。要重构表中的所有索引（或指定名称的索引），可以使用第二种形式的命令。

vacuum 通过重构数据库文件清理那些未使用的空间。如果存在处于开放状态的事务，vacuum 不会执行。一种替代手动运行 vacuum 的方法是使用 autovacuum。该功能可以通过使用 auto_vacuum 编译指示启用，下一节将会有说明。

数据库配置

120

SQLite 没有配置文件，它的所有配置参数都是用编译指示（pragma）来实现的。编译指示以独特的方式工作，有时像变量，有时像命令。它们涵盖了数据库的大部分参数，例如运行信息、数据库 schema、版本、文件格式、内存使用和调试。一些编译指示是可读的，像变量一样设置，一些要求有参数，像函数那样调用。许多编译指示既有临时形式，也有永久形式。临时形式的设置仅仅在生命期内影响当前会话。永久设置存储在数据库中影响所有的会话。缓冲区的大小就是这样的设置。

本节覆盖编译指示中最常使用的内容。您可以找出 SQLite 中编译指示的所有列表。

连接缓冲区大小

缓冲区尺寸的编译指示控制一个连接可以在内存中使用多少个数据库页。要设置当前缓冲区大小的默认值，可以按照如下方式执行：

```
sqlite> pragma cache_size;
```

```
cache_size
---------------
2000

sqlite> pragma cache_size=10000;
sqlite> pragma cache_size;

cache_size
---------------
10000
```

使用 default_cache_size 编译指示可以为所有的连接设置永久缓冲区的大小。这种设置可以存储在数据库中，且只对之后的连接生效，当前连接不受影响。

缓冲的一种用途是当连接处于预留状态（有预留锁时）时，存储那些待定的变化。正如前面一节“事务”中所描述的，如果连接填满了缓冲区，它将无法继续进一步修改，直到获得排它锁，这也意味着它可能必须等待其他读操作来清理。

如果程序对正在被许多连接使用的数据库执行很多的更新或删除操作，这将有助于增加缓冲区的大小。缓冲区越大，连接在获取排它锁之前能做的修改就越多。这样连接不仅在必须等待前可以做更多的工作，也减少了持有排它锁的时间，因为所有的工作在之前都做完了。这种情况下，持有排它锁的时间只需要足以将缓冲区的内容刷新到磁盘就可以。调整缓冲区大小的技巧将在第 5 章讨论。

获得数据库信息

使用数据库的 schema 编译指示可以获得数据库信息，定义如下：

121

- database_list：列出所有附着的数据库。
- index_info：列出索引内字段的相关信息。索引名作为参数。
- index_list：列出表中的索引信息。表名作为参数。
- table_info：列出表中所有字段的相关信息。

下面的例子说明如何使用这些参数：

```
sqlite> pragma database_list;

seq   name     file
----  -------  ---------------
0     main         /tmp/foods.db
```

```
2     db2          /tmp/db

sqlite> create index foods_name_type_idx on foods(name,type_id);
sqlite> pragma index_info(foods_name_type_idx);

seqn  cid      name
----  -------  ----------------
0     2        name
1     1        type_id

sqlite> pragma index_list(foods);

seq   name                   unique
----  -------------------    ---------------
0     foods_name_type_idx    0

sqlite> pragma table_info(foods);

cid   name           type         notn  dflt   pk
----  -------------  -----------  ----  ----   ------------
0     id             integer      0             1
1     type_id        integer      0             0
2     name           text         0             0
```

写同步

通常情况下，SQLite 会在关键时刻将所有的变化提交到磁盘以确保事务的持久性。这与其他数据库中的检查点功能类似。但是，考虑到性能因素，可以关闭这一功能，可以通过编译指示 synchronous 来实现。该编译指示有三种设置：full、normal 和 off。各自定义如下：

- full：SQLite 在继续前，在关键点暂停以确保所有的数据都实际写入磁盘。这样确保了即使操作系统崩溃或电力中断，在重启后数据库仍然是未损毁的。full 同步是非常安全的，但也较慢。
- normal：SQLite 会在绝大多数关键点暂停，但是没有 full 模式频繁。normal 模式 122
 中某个凑巧的时刻电力中断可能会导致数据库损毁的情况发生，虽然这种概率非常小。实际中，很有可能遭受严重的磁盘操作失败或者其他不可恢复的硬件故障。
- off：SQLite 将数据抛给操作系统后立即继续操作。这样可以加速一些操作，可

加速高达 50 倍。如果运行 SQLite 的应用程序崩溃，数据依然是安全的。然而，如果操作系统崩溃或计算机断电，数据库可能会损毁。

编译指示 synchronous 没有永久形式，第 5 章会讨论该参数在事务持久性的关键角色以及它是如何工作的。

临时存储器

临时存储器就是 SQLite 保存临时性数据，例如临时表、索引和其他对象的地方。在默认情况下，SQLite 使用内编译的位置，这个位置根据平台的不同而有所变化。有两个编译指示控制临时存储器：temp_store 和 temp_store_directory。第一个编译指示决定 SQLite 是使用内存还是磁盘作为临时存储器。实际上有三个可能选项：default、file 和 memory。其中，default 选项使用内编译的默认项，file 选项使用操作系统文件，memory 选项使用内存。如果将 file 设置为存储介质，那么第二个编译指示 temp_store_directory 就是存储临时文件的目录。

页大小、编码和自动清理

数据库页大小、编码和自动清理必须在创建数据库前设置。也就是说，要修改默认设置，在新数据库中创建任何新对象前都必须设置这些编译指示。默认设置继承自主机的参数，例如磁盘扇区大小、UTF-8 编码。SQLite 支持的页大小范围是 512 字节到 32,786 字节，即 2 的幂次。支持的编码有 UTF-8、UTF-16le（小字节头）、UTF-16be（大字节头）。

数据库的大小可以通过编译指示 auto_vacuum 自动保持在最小值。通常情况下，从数据库删除数据的事务提交时，数据库的大小不变。启用 auto_vacuum 时，当删除数据的事务提交时，数据库文件会缩小。为了支持这种功能，数据库内部需要存储额外的信息，这将导致数据库文件比不启用 auto_vacuum 的稍微大一些。vacuum 命令对那些使用 auto_vacuum 的数据库不起作用。

调试

有四个调试的编译指示。integrity_check 编译指示可以查看次序颠倒的记录、缺失页面、畸形记录以及损毁的索引。如果发现任何问题，返回一个描述问题的字符串。如果一切有序且正常，SQLite 返回 ok。其他编译指示可以用来追踪解析器和虚拟数据库引擎，只要数据库编译时启用了调试信息，就可以启用这些编译指示。第 11 章将会介绍关于这些编译指示的详细信息。

系统目录

sqlite_master 表是系统表，它包含数据库中所有表、视图、索引和触发器的信息。例如，foods 数据库的当前内容如下：

```
sqlite> select type, name, rootpage from sqlite_master;

type        name                 rootpage
----------  -------------------  ----------
table       episodes             2
table       foods                3
table       foods_episodes       4
table       food_types           5
index       foods_name_idx       30
table       sqlite_sequence      50
trigger     foods_update_trg     0
trigger     foods_insert_trg     0
trigger     foods_delete_trg     0
```

type 字段说明对象的类型，name 字段就是对象的名称，rootpage 指对象的第一个 B-tree 页面在数据库文件中的位置。后面的字段只与表和索引有关。

sqlite_master 表还包含称为 SQL 的字段，该字段用来存储创建对象的 DDL。例如：

```
sqlite> select sql from sqlite_master where name='foods_update_trg';

create trigger foods_update_trg
before update of type_id on foods
begin
  select case
     when (select id from food_types where id=new.type_id) is null
     then raise( abort,
                 'Foreign Key Violation: foods.type_id is not in food_types.id')
end;
End
```

查看查询计划

可以用 explain query plan 命令查看 SQLite 执行查询的方法。explain query plan 命令列出 SQLite 执行查询时访问处理表与数据的具体步骤。

使用 explain query plan 命令时，只需在该命令后面加上正常的查询文本。例如，下面的例子说明 explain query plan 给出的如何在 foods 表中执行查询。

```
sqlite> explain query plan select * from foods where id = 145;
order    from     detail
-------  -------  ----------------------------
0        0        TABLE foods USING PRIMARY KEY
```


这意味着 SQLite 正在访问表 foods，使用主键（id）执行访问（不是蛮力式的扫描表里的数据）。

研究这些查询计划是理解 SQLite 如何访问数据并满足查询的关键。您可以看到哪些地方何时、如何使用索引，以及表连接时的顺序。这对于解决那些运行时间很长的查询以及其他文件的作用是巨大的。

总结

SQL 可能是一种使用较简单的语言，但是它包含很多内容，因此，第 3、4 章也只介绍了 SQLite 实现的 SQL 中的大部分概念而已。SQL 是用户与关系数据库交互的唯一接口。不管您是系统管理员、开发者还是偶尔使用的用户，只要您与关系数据库打交道，就必须了解 SQL。

如果编程涉及 SQLite，应该去了解 SQL 其他方面的内容。现在您需要了解 SQLite 是如何执行这些命令的。这也是第 5 章要讲的内容。第 5 章会介绍 API，并展示 SQLite 函数内部是如何工作的。

第 5 章

SQLite 设计与概念

本章是后面几章的基础，后面几章侧重 SQLite 编程的不同方面，主要涉及程序员在编写代码时应该知道的 SQLite 知识。无论您是否用 SQLite 原生 C 或者脚本语言进行编程，这些知识都有助于您理解 SQLite API 和一些 SQLite 架构与实现方面的内容。掌握这些知识后，可以更好地编写代码，使得代码运行速度较快并能避免诸如死锁和难以预料的错误等潜在风险。您将看到 SQLite 如何与应用程序代码一起工作，并将确保正在正确地解决问题。

不必通过梳理源代码来了解这些内容，而且您也不必是 C 程序员。SQLite 的设计和概念非常直观，且易于理解。作为开发者，真正需要掌握的内容并不多。本章列出一些主要概念和组件用于帮助您理解 sqlite。

理解 SQLite 的关键是掌握其 API 是如何工作的。因此，本章从介绍 API 开始，讲解 API 的主要数据结构、大致设计和主要功能，也将介绍一些在查询过程中发挥重要作用的 SQLite 子系统。

除了知道函数具体做什么外，还需要从整体上纵览 API，从事务角度了解它们如何工作。SQLite 数据库的所有操作都是在事务上下文内完成的。因此，需要了解 API 背后的东西，从锁的角度理解事务是如何工作的。如果您不知道事务锁是如何运行的，它可能会给你制造麻烦。理解锁不仅可以避免潜在的并发性问题，也可以通过控制程序调用来优化查询。

最后，必须理解如何在编写代码时应用这些东西。本章最后一部分将 API、事务和锁——综合在一起进行讨论，同时，参考一些优秀的和不良的代码实例。这些实例代码给出可能导致问题的场景，并提供了一些关于如何解决这些问题的见解。

理清这些内容后，可以更好地掌握和使用 C API 及扩展语言的 API。

API

从功能上讲，SQLite API 可以分为两个主要部分：核心 API 和扩展 API。核心 API 由所有用于执行基本数据库操作的函数组成：连接数据库、处理 SQL、迭代查询结果。它还包括有助于字符串格式化、运行控制、调试和错误处理的实用函数。扩展 API 提供

通过创建用户自定义的 SQL 扩展来扩展 SQLite 的不同方式，也可以将这些自定义扩展融入 SQLite 的 SQL 语言中。

主要数据结构

正如在第 1 章看到的，SQLite 有很多组件——语法分析器、词法分析器、虚拟机等。但是从程序员的角度看，要知道的主要事情是连接、语句、B-tree 和 pager。图 5-1 显示了它们彼此之间的关系。这些对象共同解决要编写好的代码所必须知道的三个主要事情：API、事务和锁。

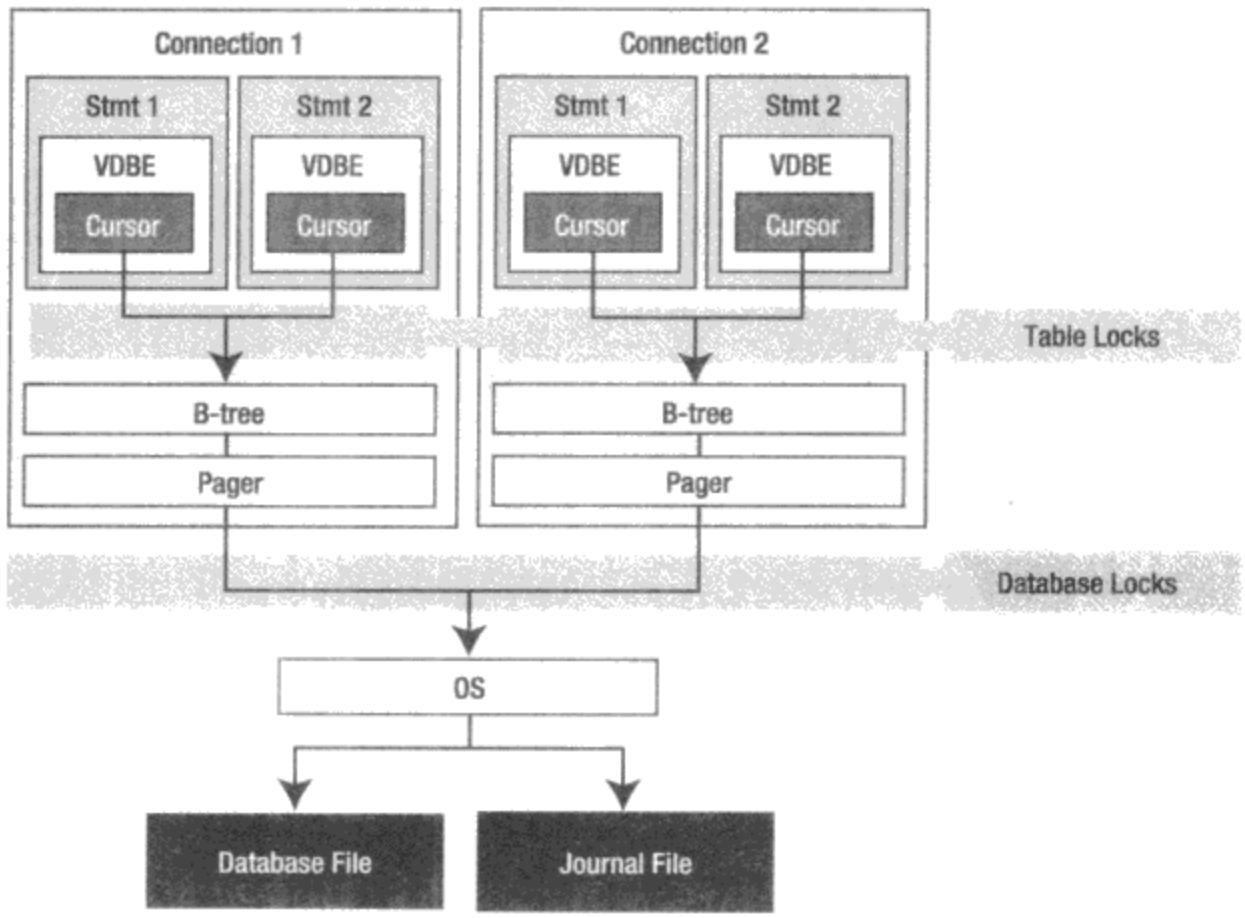


图 5-1　SQLite C API 对象模型

从技术上讲，B-tree 和 pager 并不是 API 的一部分。它们是禁地，只在 SQLite 内部使用，不对外公开。但是它们在事务和锁中发挥着重要的作用。我们将在本节和下一节“事务”中讨论它们。

连接和语句

API 中与查询处理有关的两个基本数据结构是连接和语句。在大多数扩展语言中，会看到连接对象和语句对象，它们是用来执行查询的。C API 中，它们分别对应 sqlite3 和 sqlite_stmt 句柄。API 中所有的主要操作都在使用这两个结构。

127

一个连接对象代表到数据库的连接和事务上下文，statement 来自连接对象。也就是说，每个 statement 都有一个关联的连接对象。一个 statement 代表了一个编译的 SQL 语句。在内部，它使用 VDBE 字节码来表示，VDBE 字节码是一个会执行 SQL 命令的程序。statement 包括执行一个命令所需要的一切东西。它们持有一些资源，这些资源可以保存 VDBE 程序以循序渐进的方式执行时的状态、指向磁盘上记录的 B-tree 游标和其他一些诸如边界参数的内容。我们将在“参数绑定”一节中介绍边界参数。尽管它们包含了很多不同的内容，但是可以简单地把它们理解成用来遍历结果集的游标，或者是调用 SQL 命令的不透明的句柄。

B-tree 和 Pager

一个连接对象可以连接到多个数据库对象——一个主数据库和多个附加数据库。每个数据库对象都有一个 B-tree 和相应的一个 pager 对象。

statement 使用它们连接对象的 B-tree 和 pager 从数据库中读取数据，向数据库写入数据。读取数据库的 statement 使用游标遍历 B-tree。游标遍历记录，记录存储在页面（page）内。当一个游标遍历记录时，它同时遍历页面。若一个游标想要访问页面，首先必须从内存加载到磁盘中，这是 pager 的工作。B-tree 需要数据库中的一个特定页面时，它会命令 pager 从磁盘中取出该页面。然后，pager 将该页面加载到页面缓存中，页面缓存是内存缓冲区。一旦页面在页面缓存中，B-tree 和它对应的游标就能获得页面中的记录。

如果游标修改页面，pager 必须采取特殊措施保留原始页面，确保在发生事务回滚时可以回到原来的状态。因此，pager 负责读写数据库，维护内存缓存或者页面，管理

事务。除了这些，pager 还管理锁和故障恢复。这些都会在后面的“事务”中讲述。

关于连接和事务，通常应该知道两件事情。首先，当谈到数据库的任何操作时，连接始终都只在事务里运行。其次，一个连接对象一次只能有一个打开的事务。这意味着所有来自同一个给定连接的 statement 运行在同一个事务上下文中。如果希望两个 statement 运行在不同的事务中，就必须使用多连接，每个连接对应一个事务上下文。

核心 API

之前提到过，核心 API 是用来执行 SQL 命令的，它由各种用来执行查询的函数和管理数据库的实用函数组成。执行 SQL 命令有两个必需的方法：准备查询和封装查询。在 API 和 SQLite 内部，准备查询方法最终执行所有命令的方法，它包含准备、执行和完成三个阶段，每个阶段都有一个单独的 API 函数。与执行阶段对应的是用来从结果集中获取记录和列信息的函数。

除了这些标准查询方法外，还有两种封装函数，它们将三个步骤封装成一个函数调用，并提供了一次执行 SQL 命令的简便方法。这些函数只是 API 中各种各样实用函数的一部分。我们将在这一节研究所有的查询方法和相应的实用函数。开始之前，让我们了解如何连接到数据库。

连接数据库

连接数据库比打开一个文件略微复杂一些。任何一个 SQLite 数据库都存储在操作系统文件中——一个数据库对应一个操作系统文件。C API 中用于连接或打开一个数据库的函数是 sqlite3_open()，基本上就是一个打开文件的系统调用。SQLite 也能创建内存数据库。大多数扩展中，如果使用“:memory:”或以空字符串作为数据库名，那么 SQLite 将在 RAM 中创建数据库。该数据库只能被创建它的连接访问（不能被其他连接共享）。此外，该数据库只在该连接期间存在，连接关闭时，将从内存中删除该数据库。

连接磁盘上的数据库时，数据库文件如果存在，SQLite 将打开该文件。如果不存在，SQLite 会认为您想新建一个数据库。这种情况下，SQLite 不会立即新建一个操作系统文件，只有当您向新数据库中添加内容——创建表、视图或其他数据库对象时，才会创建数据库文件。如果只是打开一个新数据库，什么都不做，然后关闭它，SQLite 不会创建数据库文件——反正它是一个空文件。

不立即新建文件有一个重要原因。某些数据库选项，如编码、页面大小以及自动清理，只能在创建数据库前设置。默认情况下，SQLite 使用的页面大小是 1024B。但是，页面大小可以设置为从 512B 到 32768B 的 2 的整数次幂的不同数值。出于性能原因，您可能想使用不同的页面大小。例如，设置页面大小与操作系统的页面大小相匹配，可以使 I/O 更加高效。较大的页面大小有助于应用程序处理大量的二进制数据。可以使用编译指示 page_size 设置数据库的页面大小。

编码是数据库的另一个永久设置。可以使用编译指示 encoding 设置数据库编码，编码可以是 UTF-8、UTF-16、UTF-16le（小端字节序）和 UTF-16be（大端字节序）。

最后是自动清理，可以使用编译指示 auto_vacuum 来设置。当一个事务从数据库中删除数据时，SQLite 的默认处理是，为了回收利用，保持被删除的页面。数据库文件大小保持不变。要释放页面，必须显式地发出 vacuum 命令来回收未使用的空间。自动清理特性可以使 SQLite 在删除数据时，自动收缩数据库文件。该功能通常在存储器比较昂贵的嵌入式应用程序中更有用。

一旦打开一个文件数据库或内存数据库——将使用一个不透明的 sqlite3 连接句柄来表示它，该句柄表示到数据库的一个连接。SQLite 扩展中，连接对象抽象了该句柄，并且有时一些对应的 API 函数实现方法将该句柄作为参数。

执行预查询

129

如前所述，预查询方法是 SQLite 执行所有 SQL 命令的实际处理方法。执行一条 SQL 命令的处理分以下三个步骤。

- 准备：语法分析器、词法分析器以及代码生成器将命令编译成 VDBE 字节码，以准备 SQL 语句。在 C API 中，这个处理由 sqlite3_prepare_v2()函数完成，该函数直接与编译器对话。编译器创建 sqlite3_stmt 句柄（语句句柄），包括：字节码、执行命令和迭代结果集所需的所有其他资源（如果命令产生结果集的话）。
- 执行：VDBE 执行字节码。执行是一个逐步的过程。C API 中，每一步都由 sqlite3_step()发起，并使 VDBE 逐步执行字节码。第一次调用 sqlite3_step()通常需要某种类型的锁，锁的类型根据执行什么样的命令（读或写）而变化。对于 SELECT 语句，sqlite3_step()的每次调用将使语句句柄的游标位置移动到结果集的下一行。对于结果集中的每一行，游标未到达结果集末尾时，将返回 SQLITE_ROW；反之，将返回 SQLITE_DONE。对于其他 SQL 语句（insert、update、delete

等），第一次调用 sqlite3_step()将促使 VDBE 执行整个命令。

- 完成：VDBE 关闭语句并释放资源。C API 中，由 sqlite3_finalize()来执行，该函数使得 VDBE 终止程序、释放资源并关闭 statement 句柄。

每一步——准备、执行、完成——对应 statement 句柄各自的状态——准备状态、活动状态、完成状态。准备状态意味着所有必需的资源已经分配，statement 已准备执行，但是没有启动，没有获得锁，也没有申请获得锁，直到第一次调用 sqlite3_step()时才开始获取锁。活动状态从第一次调用 sqlite3_step()开始。此时，statement 处于执行过程中，并且开始使用某种类型的锁。完成状态意味着开始使用关闭，并且所有相关的资源已被释放。图 5-2 显示了这些步骤和状态。

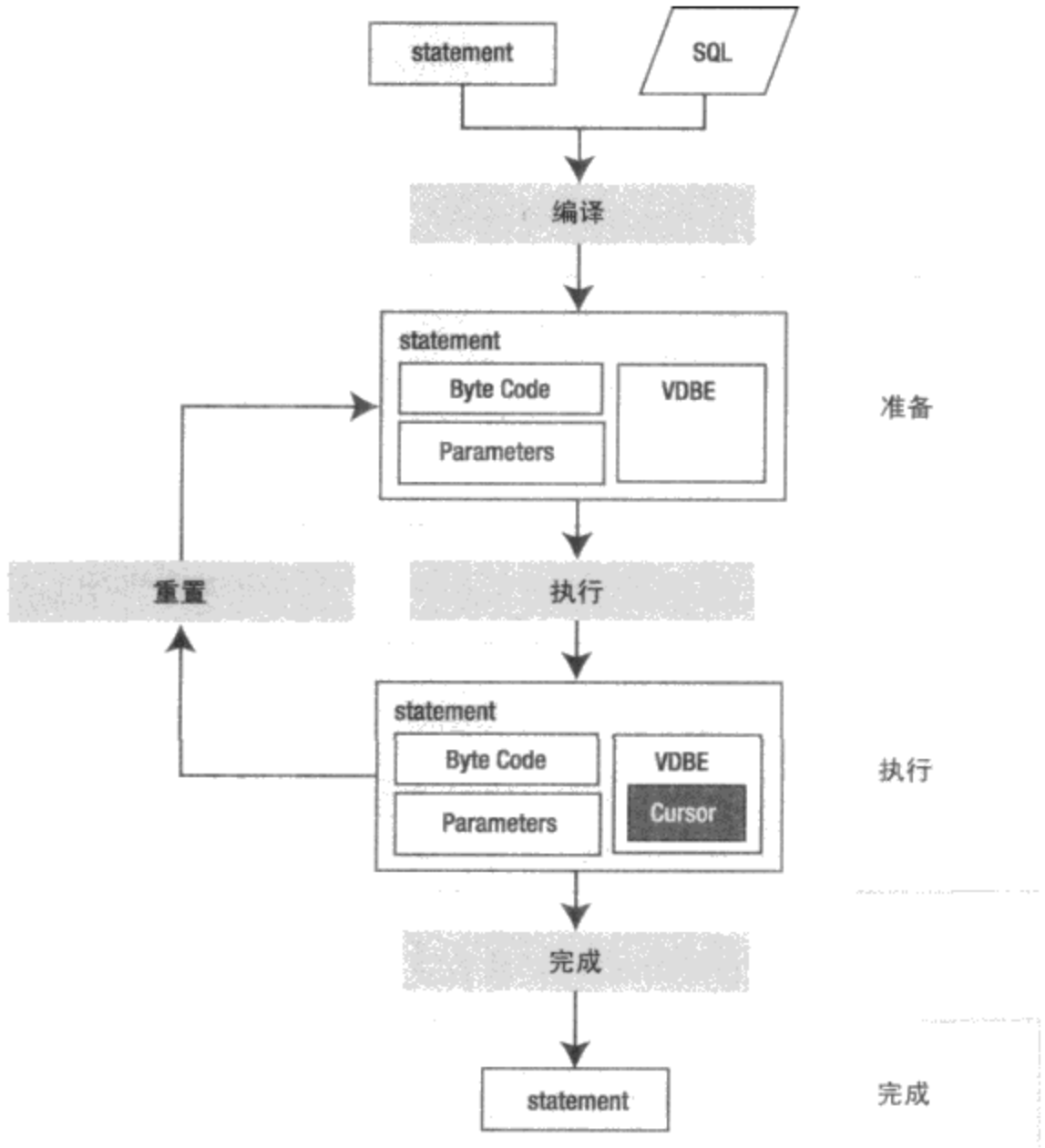


图 5-2　statement 处理

以下伪代码说明 SQLite 中执行一个查询的通用处理：

```
# 1. 打开数据库，创建一个连接对象（db）。
db = open('foods.db')

# 2.A. 准备一个 statement（stmt）。
stmt = db.prepare('select * from episodes')

# 2.B. 执行，调用 step()，直到游标到达结果集末尾。
while stmt.step() == SQLITE_ROW
  print stmt.column('name')
end

# 2.C. 完成后释放读锁。
stmt.finalize()

# 3. 插入一条记录。
stmt = db.prepare('INSERT INTO foods VALUES (…)')
stmt.step()
stmt.finalize()

# 4. 关闭数据库连接。
db.close()
```

131

这些伪代码是面向对象的 API 模拟，与您在脚本语言中看到的类似。这些方法都有对应的 SQLite API 函数。例如，prepare()对应 sqlite3_prepare()_v2，等等。这个例子执行了 SELECT，通过一个 step()调用，迭代所有返回的行，接着执行一个 INSERT。

临时存储器

临时存储器是查询处理的重要组成部分。SQLite 偶尔需要存储执行命令过程中的中间结果——例如，需要按照 ORDER BY 子句排序的结果，或一个表中的行和另一个表中的行相连接的情况。这些信息通常存储在临时存储器中。临时存储器可以是 RAM 或文件。虽然 SQLite 有适合所有平台的默认值，但您可能希望控制如何以及何处使用存储器。编译指示 temp_store 使您能够明确是否使用 RAM 或基于文件的存储。如果采用基于文件的存储，可以使用编译指示 temp_store_directory 来明确在何处创建存储文件。

使用参数化 SQL

SQL 语句可以包含参数。参数就是占位符，可能会在编译后为其提供（或“绑定”）值。下面是参数化查询示例：

```
insert into foods (id, name) values (?,?);
insert into episodes (id, name) (:id, :name);
```

这两个语句代表两种形式的参数绑定：位置和命名。第一个命令使用位置参数，第二个命令使用命名参数。

位置参数是通过语句中的问号定义的。第一个问号表示位置为 1，第二个为 2，以此类推。命名参数使用实际变量名，以冒号作为前缀。当 sqlite3_prepare_v2()编译带有参数的语句时，它在结果语句句柄中为这些参数分配占位符。然后，期望语句执行前为这些参数提供值。如果参数没有绑定值，执行该语句时，SQLite 默认使用 NULL。

132

参数绑定的优点是无须重新编译，就可以多次执行相同的语句。只需重置该语句、绑定新值，并重新执行。这时重置函数（而非完成函数）就可以发挥作用了：它可以避免 SQL 编译的开销。通过重置语句，可以重用已编译的 SQL 代码。可以完全避免词法分析、语法分析和代码生成开销。重置语句是通过执行 API 函数 sqlite3_reset()实现的。

参数绑定的另一个优点是 SQLite 会处理绑定到参数的转义字符。例如，如果参数值为“Kenny's Chicken”，参数绑定过程自动将其转换为“Kenny"s Chicken”——将单引号转义，帮助您避免了语法错误和可能的 SQL 注入式攻击（在“SQL 语句格式化”中会讲到）。下面的伪代码说明了使用参数绑定的基本过程：

```
db = open('foods.db')
stmt = db.prepare('insert into episodes (id, name) values (:id, :name)')

stmt.bind('id', '1')
stmt.bind('name', 'Soup Nazi')
stmt.step()

# 重置并再次使用
stmt.reset()
stmt.bind('id', '2')
stmt.bind('name', 'The Junior Mint')

# 完成
stmt.finalize()
```

```
db.close()
```

这里的 reset ()只释放语句资源，但是保持 VDBE 字节代码及其参数不变。该语句无须再次调用 prepare()，就可以再次运行。这可以显著提高诸如此类的重复查询的性能，因为不用重新编译。

执行封装查询

如前所述，有两个非常有用的实用函数封装了准备查询过程，使您可以在一个函数调用中执行 SQL 命令。第一个函数是 sqlite3_exec()——通常用来执行不返回数据的查询。另一个是 sqlite3_get_table()——通常用于执行返回数据的查询。许多语言扩展中，您会看到与这两个函数类似的函数。大多数扩展将第一种方法引用为 exec()，第二个方法引用为 get_table()。

函数 exec()是执行 insert、update 和 delete 语句或用于创建和销毁数据库对象的 DDL 语句的快速而简便的方法。它直接在数据库连接上工作，将打开数据库的 sqlite3 句柄连同包含一个或多个 SQL 语句的字符串作为参数。exec()能一次处理和运行以分号分隔的多个 SQL 语句字符串。

在内部，exec()解析 SQL 字符串，识别单个语句，然后一个接一个地处理它们。它分配自己的语句句柄，准备、执行并完成每个语句。如果传递给它多个语句并且其中一个失败了，exec()会在那个失败命令处终止，返回相关的错误代码。否则，它会返回一个成功的代码。下面的伪代码从概念上说明了 exec()如何以扩展方式工作：

133

```
db = open('foods.db')
db.exec("insert into episodes (id, name) values (1, 'Soup Nazi')")
db.exec("insert into episodes (id, name) values (2, 'The Fusilli Jerry')")
db.exec("begin; delete from episodes; rollback")
db.close()
```

虽然也可以使用 exec()处理 select 返回的记录，但这样做会涉及复杂的方法，通常只有 C API 支持。

第二个查询函数 sqlite3_get_table() 有点用词不当，因为它并不局限于查询单个表。相反，其名称是指表格形式的 select 查询结果。当然也可以在其中处理连接操作。get_table()与 exec()在许多方面的工作方式相同，但是它会在内存中返回完整的结果集，结果集的展示方式取决于扩展。下面的伪代码说明了通常情况下如何使用它：

```
db = open('foods.db')
table = db.get_table("select * from episodes limit 10")

for i=0; i < table.rows; i++
    for j=0; j < table.cols; j++
        print table[i][j]
    end
End

db.close()
```

get_table()的优点是它一步就可以执行查询并获得结果。其缺点是它将结果完全存储在内存中。这样，结果集越大，占用的内存就越多。因此，使用 get_table()来检索大的结果集并不合适。另一方面，准备查询方法每次在内存中只有一个记录（实际上是关联的数据库页），因此，它比较适合于获取大型结果集。

注意，虽然这些函数提供了方便，但也因为没有访问语句句柄而失去一些灵活性。例如，不能在其中使用参数化 SQL 语句。所以，对于一些会从参数化受益的重复性任务，它们的效率不高。API 还包括一些使用语句句柄提供有关结果集中的许多列信息的函数——数据和元数据，这些函数并不包括在封装查询中。封装查询有其用途，这一点可以肯定，但准备查询同样也有其用处。

错误处理

前面的示例主要用于说明查询处理的基本过程，因此，做了很多简化处理。但现实生活中，总需要考虑错误的可能性。所有您见过的函数几乎都会遇到某种类型的错误，需要处理的常见错误代码包括 SQLITE_ERROR 和 SQLITE_BUSY。其中，后一个错误代码是指繁忙情况，当连接无法获取锁时会出现繁忙情况。繁忙情况会在“事务”一节中介绍，模式错误会在第 6 章中详细介绍。

134

至于一般错误，API 通过 sqlite3_errcode()函数提供最后一个被调用函数的返回代码。可以使用 sqlite3_errmsg()函数获取更详细的错误信息，sqlite3_errmsg()函数提供最后一个错误的详细信息。大多数语言扩展以某种方式或另外的函数支持此功能。

记住，前面示例中的每个调用都应该使用类似下面的方式检查相应的错误。

```
# 检查并报告错误
if db.errcode() != SQLITE_OK
    print db.errmsg(stmt)
```

```
end
```

一般情况下，错误处理并不难。处理错误的方式取决于实际要做什么，最简单的错误处理方法与其他 API 相同——阅读正在使用的函数文档，从编码上做好防护工作。

SQL 语句格式化

您可能会看到一些扩展支持一个非常方便的函数 sqlite3_mprintf()。它是标准 C 语言库函数 sprintf()的变体。它具有特殊的替换功能，对于特定的 SQL 语句，这样替换很方便。它替代%q 和 Q%表示的位置，转义 SQL 的特定值。%q 与%s 工作方式类似，因为它取代参数列表中以 null 结尾的字符串。同时它也会将单引号字符转义，这有助于防止 SQL 注入攻击（请参阅“SQL 注入攻击”）。下面是示例：

```
char* before = "Hey, at least %q no pig-man.";
char* after = sqlite3_mprintf(before, "he's");
```

after 的值是“Hey, at least he"s no pig-man”。单引号的 he's 加倍后成为 he"s，使其成为 SQL 语句中可接受的字符串文本。%Q 格式与%q 相同，但它会用额外的单引号包围所产生的字符串。此外，如果%Q 的参数是（C 语言中）NULL 指针，它会产生没有单引号的空字符串。有关的详细信息，请参见 C API 参考附录 B 中的 sqlite3_mprintf()文档。

SQL 注入攻击

如果应用程序依赖于用户输入构造 SQL 语句，那么可能容易受到 SQL 注入攻击。如果不仔细筛选用户输入，某些人可能精心设计输入以构造出修改的 SQL 语句，在字符串中注入新的 SQL 语句。例如，假设程序使用用户输入值来填写下面的 SQL 语句：

```
select * from foods where name='%s';
```

用户所提供的输入会替换%s 。如果用户对您的数据库有所了解，他们可以提供的输入能极大地改变 SQL 语句。例如，假设用户为 name 提供以下字符串值：

```
nothing' limit 0; select name from sqlite_master where name='%
```

替换用户输入后，新的语句变成两条语句：

135

```
select * from foods where name='nothing' limit 0; select name from
sqlite_master where name='%';
```

第一条语句什么也不返回，第二条语句将返回数据库中所有对象的名称。当然，这

种情况的发生需要攻击者具有相当多的攻击知识，但是这不是不可能的。目前一些主要（商业）的 Web 应用程序将 SQL 语句嵌入在 JavaScript 中，这将提示正在使用何种数据库。前面的示例中，恶意用户只要为 sqlite_master 中的每个表执行一个 drop table 命令，您可能就要摸索如何从备份中恢复数据库了。

可操作的控制

API 中包含各种各样的使您可以监视、控制或限制数据库中发生什么的命令。SQLite 以过滤或者回调函数方式实现该功能，可以为指定事件注册要调用的函数。有三种“钩子”函数：sqlite3_commit_hook()，用于监视连接上的事务提交；sqlite3_rollback_hook()，用于监视回滚；sqlite3_update_hook()，用于监视来自 insert、update 和 delete 命令的更改操作。

■注意：即将发布的 SQLite 3.7 将包括许多新功能，其中之一是预写日志（Write Ahead Log，WAL）。为此，实现了一个新的名为 wal_hook()的钩子，它与这里讨论的钩子有些不同。第 11 章将会介绍有关 WAL 更多的详细信息。

这些钩子在运行时调用——同时执行命令。每个钩子允许您在连接上注册回调函数，并允许您提供某种形式的要传递给回调的特定应用程序的数据。操作控制函数的一般用法如下所示：

```
def commit_hook(cnx)
  log('Attempted commit on connection %x', cnx)
  return -1
end
db = open('foods.db')
db.set_commit_hook(rollback_hook, cnx)
db.exec("begin; delete from episodes; rollback")
db.close()
```

钩子的返回值有权以特定的方式改变事件，这取决于钩子的定义。本例中，因为提交钩子返回非零值，提交将被回滚。

此外，API 提供名为 sqlite3_set_authorizer()的功能非常强大的编译时钩子。该函数几乎提供对数据库中发生的一切事件的细粒度控制，并能够限制对数据库、表和列的访

问和修改。第 6 章将会详细讨论该函数。

使用线程

SQLite 拥有一批在多线程环境中使用的函数。在 3.3.1 版本中，SQLite 引入了独特的运作模式：共享缓存模式。该模式是为多线程的嵌入式服务器设计的，它可以让单个线程拥有共享页面缓存的多个连接，从而降低服务器的总内存。它还采用了不同的并发模型。该功能还包括用于管理内存和微调服务器的各种函数。稍后会在第 6 章“共享缓存模式”中进一步介绍操作模式。

扩展 API

SQLite C API 中的扩展 API 支持用户自定义函数、聚合和排序规则。用户自定义函数就是映射到用户自己用 C 语言或者其他语言实现的 SQL 函数。使用 C API 时，可以用 C 或 C++实现这样的处理程序。其他语言扩展中，可以用对应的扩展语言实现该处理程序。

用户自定义扩展必须一对一注册到连接，因为它们存储在程序内存中。也就是说，它们不像大型关系数据库系统中的存储过程，它们不存储在数据库中，而是存储在程序中。当程序或脚本启动时，为计划使用自定义扩展的连接负责注册所需的扩展。

创建用户自定义函数

实现用户自定义函数需要两步：首先，编写处理程序。该处理程序实现您要 SQL 执行的事情。其次，注册处理程序，提供 SQL 名称、参数个数以及指向该处理程序的指针（或引用）。

例如，假设想要创建名为 hello_newman()的特殊的 SQL 函数，它返回“Hello Jerry”文本。SQLite C API 中，则首先要创建一个 C 函数实现这一功能，如下所示：

```
void hello_newman(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    /* 产生 Newman 回复 */
    const char *msg = "Hello Jerry";
```

```
    /* 设置返回值 */
    sqlite3_result_text(ctx, msg, strlen(msg), SQLITE_STATIC);
}
```

如果您不了解 C 或 C API，不用担心。该处理程序只是返回“Hello Jerry”。接下来，要实际使用它，需要使用 sqlite3_create_function()注册该处理程序（或其他扩展语言中同样功能的函数）：

```
sqlite3_create_function(db, "hello_newman", 0, hello_newman);
```


第一个参数（db）是数据库连接。第二个参数是该函数的名称，该名称将出现在 SQL 中。第三个参数表示该函数的参数个数为零。（如果提供-1，意味着该函数接受数目可变的参数。）最后一个参数是指向 C 函数 hello_newman()的指针，调用 SQL 函数时将调用该指针。

注册后，当 SQLite 遇到 SQL 函数 hello_newman()时，它需要调用 C 函数 hello_newman()来获取结果。现在，可以在程序中执行 select hello_newman()命令，它将返回一行记录，其中列包含文本“Hello Jerry”。如第 4 章所述，用户自定义函数是一种简便的方法，通过将用户自定义函数嵌入 check 约束中，可以方便地实现特定领域的约束。

创建用户自定义聚合

聚合函数是那些适用于结果集中的所有记录，并从中计算某种聚合值的函数。sum()、count ()和 avg()都是 SQLite 中的标准 SQL 聚合函数。

实现用户自定义聚合需要三步：注册聚合、实现结果集中每个记录调用的步骤函数、实现结果处理后调用的完成函数。其中，完成函数允许计算最终的聚合值并执行必要的清理工作。

下面是 SQLite Python 中实现一个称为 pysum 的简单的 SUM()聚合例子：

```
connection=apsw.Connection("foods.db")

def step(context, *args):
    context['value'] += args[0]

def finalize(context):
    return context['value']
```

```
def pysum():
    return ({'value' : 0}, step, finalize)

connection.createaggregatefunction("pysum", pysum)

c = connection.cursor()
print c.execute("select pysum(id) from foods").next()[0]
```

函数 createaggregatefunction()注册聚合，传入 step()函数和 finalize 函数。SQLite 传递给 step()一个上下文环境参数，该上下文用来存储 step()调用之间的中间值。本例用于计算总和。处理完最后一条记录后，SQLite 将调用 finalize()。这里的 finalize()返回聚合的总和。SQLite 自动清理上下文环境。

创建用户自定义排序

138

排序规则定义字符串值如何比较。因此，用户自定义排序规则提供一种不同的文本比较和排序方法。这是通过 API 中的 sqlite3_create_collation()函数完成的。SQLite 提供了三种默认的排序规则：BINARY、NOCASE 和 RTRIM。BINARY 使用 C 函数 memcmp()（它是区分大小写的）比较字符串值。NOCASE 刚好相反——它的排序是不区分大小写的。RTRIM 排序规则的工作方式与 BINARY 类似，不同之处在于它将忽略尾随空格。

对于那些默认的 BINARY 排序规则不能很好地支持的语言和那些需要支持 utf-16 的语言而言，用户自定义排序非常有用。对于特定的应用程序，用户自定义排序也是有用的，例如，那些不用字典顺序和时间顺序排列的日期格式排序。第 7 章会实现一个用户自定义排序规则，在 SQLite 中对 Oracle date 排序。

事务

现在您应该对 API 布局有一个整体的认识。了解了执行 SQL 命令的不同方式，以及一些有用的实用程序函数。但是，SQL 执行命令远比 API 中的内容多。事务和锁在查询处理中密切相关。查询总是在事务内执行，事务又涉及锁，如果锁控制不当，会导致很多问题。您可以通过 SQL 和代码控制锁的类型和持续时间。

第 4 章借助两个只通过 SQL 管理事务的连接例子，说明引起死锁的具体场景。程序员会用另一个变量——代码——来判断在任何给定时间的多个连接，这些连接有多个

状态与多个语句。所有的方法都是单个语句句柄，甚至您有时都没有意识到，您的代码可能持有独占锁，进而阻止了其他连接做任何事情。

这就是为什么掌握好事务和锁的工作方式以及它们如何与各种 API 函数关联以执行查询很重要。理想情况下，您应该能够看懂自己写的代码并能指出事务的状态，或者至少能够发现潜在的问题。本节中，我们将探索事务和锁背后的机制，并在下一节观察它们如何在实际的代码中工作。

事务生命周期

代码和事务要考虑一些事情。首先要知道事务在哪些对象上运行。接着就是持续时间问题——事务何时开始和何时结束，在什么时候开始影响其他连接？第一个问题与 API 直接相关。第二个问题一般与 SQL 和 SQLite 的特定实现相关。

139

如您所知，单个连接可以分配多个语句句柄。如图 5-2 所示，每个连接都有一个 B-tree 和与之关联的 pager。pager 比连接的作用大，因为它管理事务、锁、高速缓存和故障恢复——所有这些内容都将在接下来的几节中有所涵盖。可以简单地说，连接对象处理这一切，但实际上，是 pager 处理这一切。要记住的重要一点是，数据库写操作时，是在用一个连接，一次一个事务。因此，所有语句的对象都是运行在派生它们自身的连接的单个事务上下中。这回答了第一个问题。

对于第二个问题，事务持续时间（或事务生命周期）与单个语句一样短暂，或者，只要您喜欢——也可以很长，直到喊停时才停止。默认情况下，连接运行在自动提交模式下，这意味着发出的每个命令都运行一个单独的事务。相反，当发出 begin 命令时，事务将持续到调用 COMMIT 或 rollback，或者 SQL 命令引起约束违反进而导致 rollback。接下来的问题是事务如何与锁关联。

锁状态

大多数情况下，锁持续时间隐藏在事务持续时间内。虽然这两个并不总是一起开始，但是它们总是一起结束。事务结束时，释放与其相关的锁。另一种说法是，只有与锁关联的事务结束时，或者在更糟的情况下，程序崩溃时，才会释放锁。如果程序或系统崩溃，并且这种情况下事务没有结束，那么数据库依然有隐式锁，并且只能通过访问它的下一个连接来解决。稍后的“锁与故障恢复”会介绍。

SQLite 有 5 种不同的锁状态，连接总是处于其中之一。图 5-3 显示了 SQLite 的锁状态和转换，该图详细列出了连接可能处于的所有的锁状态，并给出了生命周期中可以转换的所有路径。该图是通过锁状态、锁转换和事务生命周期说明的，图中可以看到事务中的锁状态。

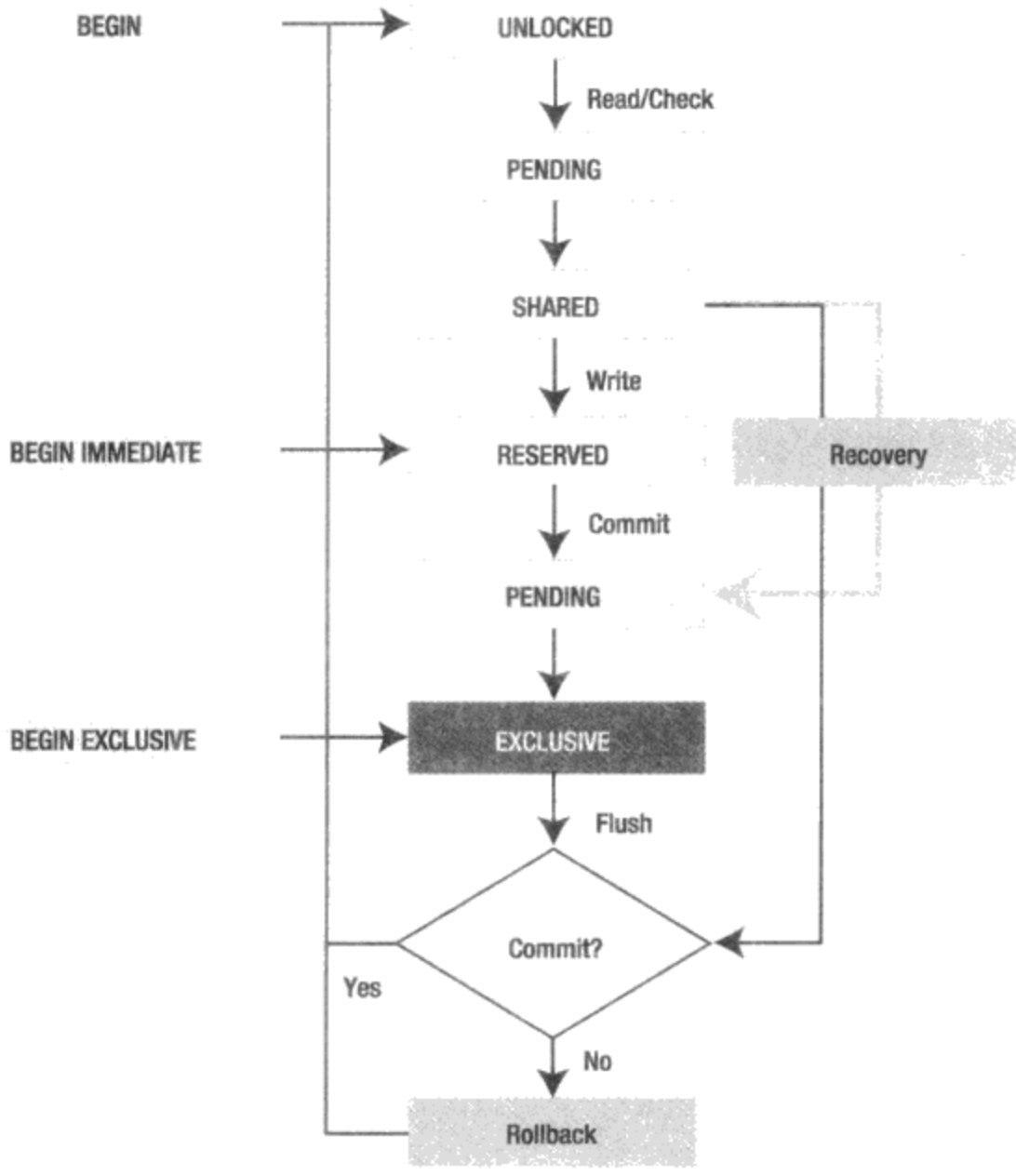


图 5-3　SQLite 锁转换

每个状态都有一个相应的锁，未锁定除外。因此，可以说连接“有保留锁”或“处于保留状态”，或只是“在保留”，它们的意思都相同。除了未锁定外，处于给定状态的连接必须首先获得相关联的锁。

所有的事务都是从未锁定、保留或排他锁开始的。如图 5-3 所示，默认情况下，一

切都从未锁定开始。白色的锁状态——未锁定、待定、共享和保留——全都可以在同一时间同一数据库的不同连接中存在。不过，从灰色的待定锁开始，限制就更多了。灰色的待定状态代表锁正在被某个连接拥有，即某个想要获取独占锁的写操作。与此相反，白色的待定状态表示连接获取和释放共享锁的途径。尽管有这些不同的锁状态，但是所
141 有的 SQLite 事务都可以归结为两种类型之一：读事务和写事务。这也是锁图描绘的最终内容：读操作、写操作以及它们如何在一起工作。

读事务

从 select 语句的锁进程开始，它的路径非常简单。执行 select 语句的连接启动事务，从未锁定转到共享锁，提交之后回到未锁定状态，操作结束。

现在，您也许会很好奇：执行两个语句会发生什么？它们的锁路径是什么？这取决于是否运行在自动提交模式下。考虑如下示例：

```
db = open('foods.db')
db.exec('begin')
db.exec('select * from episodes')
db.exec('select * from episodes')
db.exec('commit')
db.close()
```

这里有明确的 begin 命令开始，两个 select 命令在一个事务中执行，因此，它们在同一个共享状态下执行。第一个 exec()运行，让连接进入共享状态；然后第二个 exec()运行；最后，手动提交命令，让连接从共享状态回到未锁定状态。代码的锁路径如下所示：

```
UNLOCKED→PENDING→SHARED→UNLOCKED
```

现在考虑没有 begin 和 commit 命令的情况。两个 select 命令运行在自动提交模式下。因此，它们各自经历完整的路径。现在，代码的锁路径如下所示：

```
UNLOCKED→PENDING→SHARED→UNLOCKED→PENDING→SHARED→UNLOCKED
```

由于该代码只是读取数据，因此，可能不会产生多大差异，但在自动提交模式下的确会两次锁定文件，而不是其他方式中那样只锁定一次。您会看到，可以偷偷在两个 select exec()调用之间插入修改数据库的写操作，因此，无法确保两个命令返回相同的结果。另一方面，如果有 begin…commit 命令，可以保证它们的结果完全相同。

写事务

现在我们认识一下数据库写操作，例如 update 语句。首先，连接必须遵从与 select 相同的路径，先到共享状态。所有的操作——写操作或者读操作——都必须经历：未加锁→待定锁→共享锁。您很快就会看到，待定锁就是网关锁。

保留状态

连接尝试向数据库写入内容时，必须从共享锁转换到保留锁。如果它获得保留锁，则准备好开始进行数据修改。即使连接真的不能在此时修改数据库，它也可以将修改内容存储在本地 pager 内的内存缓存中，也就是前面提到的页面缓存。此缓存就是前面第 4 章中提到的可以通过编译指示 cache_size 配置的缓存。

142

当连接进入保留状态时，pager 初始化回滚日志。回滚日志是一个文件（如图 5-1 所示），用于回滚和故障恢复。具体地说，它拥有将数据库还原到事务开始之前的原来状态的数据库页。当 B-tree 修改页时，pager 将这些数据库页都存放到日志文件。本例中，对 update 修改的每条记录，页面获取与原始记录相关的数据库页，并将它们复制到日志中。日志就拥有事务开始之前的一些数据库内容。因此，要撤销事务时，pager 只是简单地将日志文件中的内容复制回数据库中。这样，数据库就还原到事务开始前的状态。

保留状态下，pager 实际上管理三种页：已修改页、未修改页和日志页。已修改页是包含 B-tree 已改变记录的页，这些页存储在页缓存中。未修改页是 B-tree 读取但并未改变的页，它们是诸如 select 命令之类的结果。最后是日志页，它就是已修改页的原始版本。日志页不会存储在页面缓存中，但 B-tree 修改前会将其写入日志。

因为页面缓存，写操作连接的确可以在保留状态完成实际的工作，而不用干扰其他（读操作）连接。因此，SQLite 可以有效地让多个读操作和一个写操作同一时间在同一数据库中工作。唯一需要注意的是，写操作连接要将所做的修改存储在页面缓存中，而不是数据库文件中。此外要注意，给定数据库同一时间只能有一个保留或独占连接——但是多个读操作可以和一个写操作并存。

待定状态

当连接完成 update 操作，并提交事务时，pager 开始进入独占状态的过程。在第 4 章中，您已经知道如何工作了，但为了完整性，这里会重复。pager 试图从保留状态获

取待定锁。一旦获得待定锁，并继续持有该锁，阻止其他连接获取待定锁。参考图 5-3 就知道这样有何作用。我们曾提到过待定锁是网关锁。现在您知道原因了，因为写操作继续持有待定锁，其他连接无法从未锁定转换到共享状态，结果是没有可以进入数据库的新连接：没有新的读操作，没有新的写操作。挂起状态实际上是损耗阶段。写操作保证它可以排队等待数据库——只要每个人都守规矩且行为得当，最终都可以获得独占锁。只有其他已有共享锁的连接可以继续正常工作。待定状态下，写操作等待这些连接完成并释放其共享锁。等待锁涉及另一个问题，稍后会在“等待锁”中讨论。

当其他连接释放了锁，数据库就属于写操作了。然后，pager 从待定状态转换到独占状态。

独占状态

独占状态中，主要工作是将修改的页从页面缓存刷新到数据库文件。这时要慎重，因为 pager 开始实际修改数据库了，对此需要格外小心。

在 pager 开始写入修改的页前，首先要处理日志。它会检查日志的完整内容是否已写入磁盘。这种情况下，依然很可能，即使 pager 已将页写入日志文件，但是操作系统
143 具有很多缓存，因此，可能还有一部分内容在操作系统的缓存中，即使不是全部。pager 告诉操作系统，将所有的页逐字写入磁盘。如第 4 章中所述，编译指示 synchronous 可以在这里发挥作用。设置同步方法确定 pager 是如何确保操作系统将日志页提交到磁盘的。正常设置是在继续之前，执行单个的“sync”告诉操作系统确认将所有缓冲的日志页写入磁盘。如果将 synchronous 设置为 FULL，在继续处理前，pager 执行两次同步操作，确保将日志提交到磁盘。如果将 synchronous 设置为 NONE，pager 根本不在乎日志（虽然速度可以快 50 倍，但是也将和事务持久性拜拜）。

将日志提交到磁盘之所以非常重要，是因为如果程序或系统在 pager 写入数据库文件时崩溃，日志是以后还原数据库文件的唯一方式。如果在系统崩溃前，日志页没有完整地写入磁盘，那么数据库就无法完全还原到其原来的状态，因为内存中的日志页在系统崩溃时丢失了。这种情况下，运气好的话，还有一个处于不一致状态的数据库，更糟的话，数据库损坏了。

■**注意**：即使设置了最保守的 synchronous，仍然可能无法保证日志真实提交到了磁盘。这不是 SQLite 的缺陷，而是某些类型的硬件或者操作系统的问题。SQLite 在 UNIX 上使用 fsync() 系统调用，在 Windows 上使用 FlushFileBuffers(),

强制将日志页刷新到磁盘。但根据报道，这些函数并不总是可以正常工作，特别是在廉价的 IDE 磁盘上。显然，一些 IDE 磁盘制造商使用了会歪曲数据提交到磁盘真相的控制器芯片。某些情况下，芯片在其易变内存中缓存了数据，但是却发出数据已写入驱动器的报告。此外，还有（未经证实）的报告，Windows 偶尔会忽略 FlushFileBuffers()。如果硬件或软件欺骗了您，事务可能不如您想象的那样持久。

一旦处理完日志，pager 就可以将所有已修改的页复制到数据库文件。下一步做什么取决于事务模式。如果在本例这种情况下，事务自动提交，然后 pager 清理日志，清除页缓存，从独占锁回到未锁定状态。如果该事务未提交，pager 继续持有独占锁，日志继续发挥作用，直到发出 COMMIT 或者 ROLLBACK 命令。

自动提交与效率

考虑一下运行在显式事务和自动提交模式下的 update 有什么不同。自动提交模式下，所有修改数据库的命令都运行在单独的事务中，并通过以下路径：

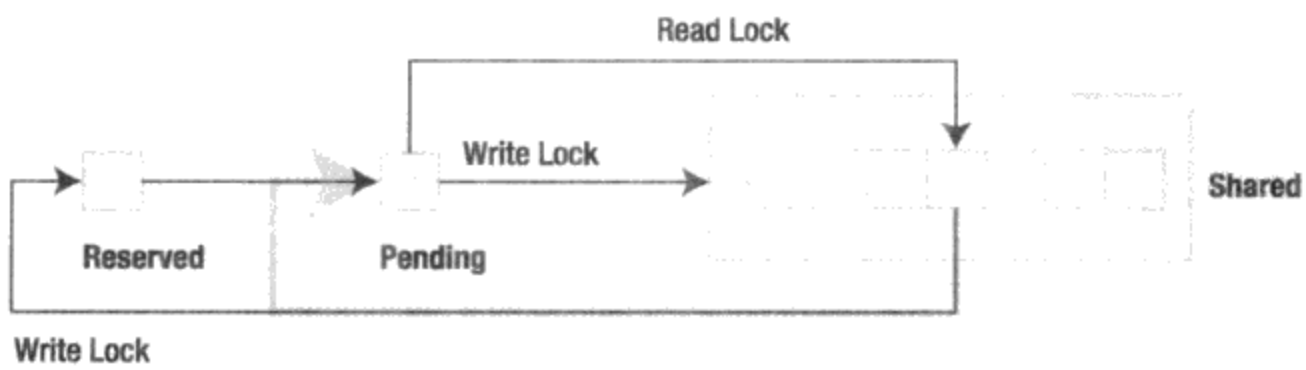
```
UNLOCKED→PENDING→SHARED→RESERVED→PENDING→EXCLUSIVE→UNLOCKED
```

此外，通过此路径的每个行程都涉及创建、提交和清除回滚日志。在自动提交模式下运行多个 select 并不能提高性能，应该为频繁的写操作重新考虑一下自动提交模式。正如前面提到的 select 的例子，在自动提交模式下运行多个命令时，不能在命令执行期间阻止其他连接修改数据库。这也是，如果您有两个依赖于数据库中特定值的 update 操作，应该始终在同一事务中运行它们的原因。

144

锁定与故障恢复

SQLite 的锁是基于标准的文件锁定实现的。SQLite 在数据库文件中有三种不同的文件锁：保留字节、待定字节和一个共享区域。



一切从待定字节开始。若要从未锁定移动到共享，连接首先尝试在待定字节上获取读锁。如果成功，在共享区域的任意字节上获得读锁，并释放待定字节上的读锁。若要从共享状态转移到保留状态，连接尝试获取对保留字节上的写入锁。若要从保留状态转换到独占状态，连接尝试获得待定字节上的写入锁。如果成功，因为其他连接不可能获取待定字节上的读取锁而进入共享状态，因此称为消耗处理。最后，若要获取独占锁，连接尝试获取整个共享区域的写入锁。由于共享区域持有其他活动连接的读取锁，此步骤保证了只有其他共享锁释放后，才能授予独占锁。

SQLite 的故障恢复机制使用保留字节来判断什么时候需要恢复数据库。需要联合使用日志文件和保留锁，如果 pager 看到前者，而没看到后者，肯定有问题发生。pager 每次打开一个数据库文件，或尝试从数据库中读取页时，都要执行简单的一致性检查。如果它发现日志文件，但是数据库中没有保留锁，那么创建日志文件的进程肯定崩溃了，或者系统死机了。这种情况下，日志被称为热日志，数据库可能潜在的处于不一致的状态。若要让一切正常，需要“回放”日志，将数据库还原到初始的被中断的事务之前的状态。

若要回放，pager 要将数据库置于恢复模式。为此，它要按图 5-3 中灰线的说明直接从共享状态转移到待定状态，这是唯一的一次这种过渡。跳过保留锁的原因有两个。首先，通过锁定待定字节，可以将所有的新连接排除在数据库外。第二，（共享）数据库中已有的连接，下次试图访问页时也会看到热日志。这些连接将尝试进入恢复模式和回放日志。但是，它们不一定能这样做，因为第一个连接已经有待定锁。因此，从共享状态进入待定状态，第一个连接确保（1）没有任何新的连接可以进入数据库，（2）处于共享状态的活动连接不能进入恢复模式。除了恢复连接外，所有的连接都暂时待定。

145

基本上，热日志就是隐式的独占锁。如果一个写操作崩溃，只有当数据库恢复后才可以有其他操作。下一个 pager 会看到热日志、锁定其他连接，开始恢复。如果没有其他活动连接，连接到该数据库的第一个程序将检测热日志并开始恢复。

调整页面缓存

如果我们更改前面的示例，并且假设从 begin 命令开始，紧接着是 update，运行中间让所做的修改填满页面缓存（运行内存不足），SQLite 会如何处理？也就是说，update

导致的修改页面填满了页面缓存会发生什么？

过渡到独占状态

真正的问题是，什么时候 pager 从保留锁转移到独占锁？为什么？有两种情况，就是您刚刚看到的那两种。要么连接到达提交点，必须进入独占状态，要么页面缓存已满并且没有其他选择。我们刚看了到达提交点的情况。所以，当页面缓存已满时会发生什么？简单地说，pager 不能再存储任何已修改的页，并因此不能做任何工作。为了继续工作，它被迫进入独占状态。实际上，这不完全是真的，因为有软限制和硬限制。

软限制对应页面缓存的第一次填满。此时，缓存是已修改和未修改页的混合。这种情况下，pager 尝试清除页缓存。它逐页遍历缓存，将未修改页清除出去。一旦这样做后，当缓存再次填满，它就再次释放内存。它将重复该过程，直到缓存完全由修改过的页填满。而此时就是硬限制，pager 此时没有其他资源，只能进入独占状态。

处于保留状态时，编译指示 cache_size 会导致差别。如第 4 章中所述，cache_size 控制页面缓存的大小。页缓存越大，pager 可以容纳的修改页越多，连接在必须进入独占状态前做的工作也越多。此外，前面提到，通过在保留状态完成所有数据库的工作，可以最小化独占状态的时间。如果在保留状态完成所有的工作，那么独占状态的持有时间只是将修改页面刷新到磁盘那么长——不用编译更多的查询，处理更多的结果，然后写到磁盘。在保留状态中处理，可以显著提高整体的并发性。理想情况下，如果您有大量的事务或拥挤的数据库，您可以空出一些内存，以确保高速缓存的大小尽可能长时间地容纳处于保留状态的连接。

调整页面缓存

如何确定高速缓存的大小？这取决于您在做什么。假设要更新 episodes 表的所有记录。这种情况下，您知道表中的每一页都将被修改。因此，您计算 episodes 中有多少个页面，并据此调整高速缓存的大小。可以使用 sqlite3_analyzer 获取 episodes 上的所有信息，该函数会转储出表中详细的统计信息，包括总页数。例如，如果在 foods 数据库中 146
运行它，可以获得关于 episodes 的以下信息：

```
*** Table EPISODES ***************************************************
```

```
Percentage of total database. ·····20.0%
Number of entries ·····181
Bytes of storage consumed ·····5120
Bytes of payload ·····3229      63.1%
Average payload per entry ·····17.84
Average unused bytes per entry ·····5.79
Average fanout ·····4.00
Maximum payload per entry ·····38
Entries that use overflow ·····0        0.0%
Index pages used ·····1
Primary pages used ·····4
Overflow pages used ·····0
Total pages used ·····5
Unused bytes on index pages ·····990      96.7%
Unused bytes on primary pages ·····58       1.4%
Unused bytes on overflow page ·····0
Unused bytes on all pages ·····1048     20.5%
```

总页数为 5，但表实际只使用了 4 页——第 1 页是索引。由于默认的高速缓存大小是 2000 页，所以不用担心。表 episodes 大约有 400 条记录，也就是每页约有 100 个记录。执行 update 命令修改所有的记录前，不必担心调整页面缓存，除非表 episodes 至少有 196,000 行记录。记住，只有在其他连接也在使用数据库并且需要考虑并发的环境中才需要调整页面缓存的大小。如果只有一个连接使用该数据库，就不需要调整。

等待锁

我们刚才说过 pager 从待定状态到独占状态要等待。具体涉及锁的哪些内容？首先，任何对 exec()或 step()的调用都会涉及等待锁。每当 SQLite 遇到不能获取锁的情况时，默认行为是向导致该获取锁行为的函数返回 SQLITE_BUSY。不管执行什么命令，都可能潜在遇到 SQLITE_BUSY。如您所知，如果写操作正在待定或者正在写，SELECT 命令将无法获取共享锁。遇到 SQLITE_BUSY 时的简单处理就是重新再调用一次。但是，我们很快会看到这并不总是最好的解决方法。

繁忙处理

不用反复重试该调用，可以使用一个繁忙处理程序。如果连接无法获取锁，不用让

API 返回 SQLITE_BUSY，而是让它调用繁忙处理程序。

繁忙处理程序是您创建的消磨时间或任何其他您希望它做的函数——它可以向所有关心 SQLite 的人发送垃圾邮件。SQLite 无法获取锁时，就会调用它。繁忙处理程序唯一要做的事情就是提供一个返回值，告诉 SQLite 接下来做什么。按照约定，如果该处理程序返回 true，SQLite 会继续获取锁。如果返回 false，SQLite 将向请求锁的函数返回 SQLITE_BUSY。参考如下示例：

147

```
counter = 1

def busy()
    counter = counter + 1
    if counter == 2
        return 0
    end

    spam_mother_in_law(100)
    return 1
end

db.busy_handler(busy)
stmt = db.prepare('select * from episodes;')
stmt.step()
stmt.finalize()
```

spam_mother_in_law()的实现留给读者作为练习。

执行 select 操作，step()函数必须获取数据库的共享锁。但是，假设有一个处于活跃状态的写操作。通常情况下，step()会返回 SQLITE_BUSY。然而，在本例中它不会这样。pager（它负责处理锁）会调用 busy()函数，因为已注册了繁忙处理程序。busy()的计数器加一，将随机消息 100 转发，并返回 1，pager 将 1 解释为 true——不断尝试获取锁。然后 pager 再次尝试获取共享锁。假设数据库仍处于锁定状态，pager 再次调用繁忙处理程序。这一次，busy()返回 0（因为计数器等于 2 了），pager 将 0 解释为 false。在这种情况下，不再重试获取锁，pager 向堆栈发送 SQLITE_BUSY，这也是 step()结束的返回值。

如果您只是想消磨时间来等待锁，您就不必编写繁忙处理程序，SQLite API 自身就有一个，它在一个给定的时间内等待锁。在 API 中，该函数是 sqlite3_busy_timeout()，

某些扩展库也支持它。实质上，它相当于说“如果您不能获取锁睡上 10 秒钟”，pager 会做这件事。如果已经休眠 10 秒钟，仍然不能获取锁，它将返回 SQLITE_BUSY。

使用恰当的事务

再次考虑前面的例子，这一次是 update 命令，而不是 select 命令。SQLITE_BUSY 实际上意味着什么？在 select 中，它只意味着“我不能获取共享锁。”但在 update 中，它意味着什么呢？事实是，您真的不知道它的意思。SQLITE_BUSY 可能意味着连接无法获取共享锁，因为有挂起的写操作。它也可能意味着它有共享锁，但是无法转换到保留锁。问题是在这种情况下不知道数据库或者连接的状态。自动提交模式下，SQLITE_BUSY 对于写操作完全是不确定的。所以，下一步该做什么？应该继续调用 step()直到命令通过吗？

这是需要考虑的事情。假设 SQLITE_BUSY 是获取共享锁而不是保留锁的结果，正在拥有一个从保留锁到独占锁的连接。其次，不知道数据库的状态，一味用蛮力推进事务不一定会有效。如果持续调用 step()，可能因为其他拥有保留锁的连接而无法继续，如果那些连接不让步的话，将进入死锁。

注意：SQLite 通过忽略违法连接的繁忙处理程序来试图帮助这种特定情况下的死锁预防。如果它正在防止保留锁，那么就不触发共享锁连接的繁忙处理。但是，由代码负责提示。代码依然可以反复调用 step()，在这种情况下，SQLite 就帮不上什么忙。

因为您要进行数据库写操作，需要发出 begin IMMEDIATE 命令。如果得到 SQLITE_BUSY，那么至少您知道在什么状态。知道您可以安全地继续尝试而不会阻挡另一个连接，最终一旦成功，您知道自己处于什么状态——保留锁。现在，如果必须的话，可以使用蛮力，因为您是当前唯一正确的连接。如果使用 begin exclusive 启动，另一方面，可以保证您不会有要处理的任何繁忙的情况。只需要记住这种情况下您正以独占方式工作，与保留锁相比，这对并发而言并不好。

锁与网络文件系统

这种情况下，您应该理解网络文件系统共享的数据库有时出错的情况。SQLite 通过对数据库文件加锁来处理并发性。非常重要的一点是在正确的时间加锁或释放锁。

SQLite 完全依赖于文件系统管理并发使用的锁。无论是在正常的文件系统中还是在网络文件系统中运行，SQLite 均使用相同的锁定机制。它在 UNIX、Linux OS X 上使用 POSIX 建议锁，在 Windows 上使用 LockFile()、LockFileEx()、UnlockFile()系统调用。这些调用是标准的系统调用，在正常的文件系统上能正常工作。网络文件系统模拟正常文件系统工作。不幸的是，某些网络文件系统的实现并不总是能正确地模拟正常的文件系统。即使网络文件系统能正常工作，还有其他要考虑的因素。

以 NFS 为例，它是一个很好的网络文件系统。不过，已知许多最初的 NFS 实现都有缺陷，或在一些情况下未实现锁定。与 SQLite 数据库工作时，这可能导致严重问题。没有锁定，两个连接可以在同一数据库上获取独占锁，同一时间可以写数据库，几乎必然导致某些数据库损坏。这不是 NFS 协议问题，而是该协议的一些特定实现的问题。值得庆幸的是，绝大多数现代化的实现都解决了这些问题——一些实现，例如 SUN 公司的——锁定实现坚如磐石。

如果某个客户端死机（和它们的数据库一起），NFS 仍会有问题，并且无法恢复。例如，NFSv3 中被锁定的文件需要管理员权限的干预才能清理锁。NFSv4 中，设定一个超时时间。但超时时间多长？三十秒？一分钟？真正的问题不是在 NFS，而是特定的应用程序。网络文件系统即使实现得很完善，也不一定会与本地文件系统完全一样。

如果想将网络文件系统和并发同时使用，即使使用最好的网络文件系统，还是有许多要考虑的问题。这并不像在本地文件系统中运行那么简单。 149

代码

现在，对 API、事务和锁已经有了全面的了解。让我们把这三个内容联系起来，放在如下的代码中，您可能想看如下几个方案。

使用多个连接

如果您写过使用其他关系数据库的代码，您可能在一个代码块中使用多个连接。典型的例子是一个连接遍历表，而另一个连接修改记录。

SQLite 中，一个代码块中使用多个连接可能会出现问题，处理它们时需要特别注意。

考虑如下示例：

```
c1 = open('foods.db')
c2 = open('foods.db')

stmt = c1.prepare('select * from episodes')

while stmt.step()
  print stmt.column('name')
  c2.exec('update episodes set <some columns, criteria, etc.>)
end

stmt.finalize()

c1.close()
c2.close()
```

相信您可以轻松地找出这里的问题。在 while 循环中，c2 试图执行更新操作，此时 c1 的共享锁依然打开。直到 while 后 stmt 才会完成，这时共享锁才会释放。因此，在 while 循环期间是无法对数据库执行写操作的。更新操作或者以静默方式失败，或者如果您有一个繁忙处理程序，可以将更新操作延迟。这里最好使用一个连接，并运行在 begin immediate 开始的事务下。新版可能如下所示：

```
c1 = open('foods.db')

# 继续尝试，直到返回 SQLITE_SUCCESS
while c1.exec('begin immediate') != SQLITE_SUCCESS
end

stmt = c1.prepare('select * from episodes')

while stmt.step()
  print stmt.column('name')
  c1.exec('update episodes set <some columns, criteria, etc.>)
end

stmt.finalize()
c1.exec('commit')
c1.close()
```

这种情况下，应该使用单个连接进行读取或写入，不必担心数据库锁定导致的问题。

然而，事实证明，这个例子仍然无法工作。如果您用一个语句遍历表的同时用另一个语句更新表，则会有额外的锁定问题，您需要知道这些，我们将在下面讨论。

finalize()函数的重要性

处理 select 语句的常见疑问是错误地认为，直到调用 finalize()（或称为 reset()），才必须释放共享锁——大部分情况下是这样。请考虑如下示例：

```
stmt = c1.prepare('select * from episodes')

while stmt.step()
  print stmt.column('name')
end

c2.exec('begin immediate; update episodes set …; commit;')

stmt.finalize()
```

虽然永远不应该在实践中这样做，但是您可能这样做了，并且可能侥幸成功。如果用 C API 编写同样的程序，实际上是可以工作的。即使没有调用 finalize()，第二个连接还是可以修改数据库的。之前已经讨论过原因，请看如下示例：

```
c1 = open('foods.db')
c2 = open('foods.db')

stmt = c1.prepare('select * from episodes')

stmt.step()
stmt.step()
stmt.step()

c2.exec('begin immediate; update episodes set …; commit;')

stmt.finalize()
```

假设 episodes 有 100 条记录。程序逐步向前，只通过了其中的三个，这时会发生什么？第二个连接将获得 SQLITE_BUSY。 151

第一个示例中，当语句到达结果集末尾时，SQLite 释放了共享锁。也就是在最后调用 step()，其中 API 返回 SQLITE_DONE，VDBE 遇到 Close 指令，SQLite 关闭游标并

放弃共享锁。因此，即使 c1 不调用 finalize()，c2 依然能够执行插入。

第二个示例中，该语句尚未到达结果集末尾。下一个 step()调用会返回 SQLITE_RESULT，这就意味着结果集中有很多行并且共享锁仍处于活动状态。因此，由于来自 c1 的共享锁，c2 不能执行插入操作。

上述示例的寓意就是让大家不要这样做，即使有时可以行得通。您在写另一个连接之前始终要调用 finalize()或 reset()。要记住的另外一点是在自动提交模式下，step()和 finalize()或多或少是事务和锁定的边界。它们开始事务，结束事务。它们获取锁，然后释放锁。在这些函数调用之间处理其他连接时，应该特别注意。

共享缓存模式

现在，明确了并发性规则后，又有了一些新的困惑。SQLite 提供了另一种并发模式：共享缓存模式，它是关于单个线程中多个连接如何操作的。

在共享缓存模式中，一个线程可以创建共享相同页面缓存的多个连接。此外，该组连接可以同一时间在同一数据库中有多个读操作和一个写操作（独占的）工作。困难在于这些连接不能在线程间共享——它们被严格限制在创建它们的线程内（特别是在共享缓存模式下运行）。此外，读和写必须准备好处理涉及表锁的特殊情况。

读表时，SQLite 自动加上表锁，这样可以防止写操作修改这些表。如果写操作试图修改被读操作锁定的表，它将得到 SQLITE_LOCKED。读操作试图读取被写操作锁定的表遵从相同的逻辑。不过，在后一种情况中，只要运行在读取未提交模式下，读操作仍然可以继续，并读取正在被写操作修改的表，可以通过编译指示 read_uncommitted 启动读取未提交模式。这种情况下，SQLite 不会在读取的表上加读锁定。因此，这些读操作不干涉写操作。不过，这些读操作可能得到不一致的查询结果，因为写操作在读取表的时候可以修改表。这与您可能在其他关系数据库中看到的“读脏数据”的概念类似。

共享的缓存模式是为某种情况下需要保留内存同时有稍高并发的嵌入式服务器设计的。第 6 章中，可以找到更多共享缓存模式使用 C API 的详细信息。

总结

SQLite API 是灵活、直观，而且易于使用的。它由两个基本部分组成：核心 API 和

扩展 API。核心 API 围绕两个基本的用于执行 SQL 命令的数据结构：连接和语句。命令通过三个步骤执行：编译、执行和完成。SQLite 的封装函数 exec()和 get_table()将这三步封装成一个函数调用，自动处理相关联的语句对象。API 扩展提供了三种不同的自定义 SQLite 的方式：用户自定义函数、用户自定义聚合和用户自定义排序规则。

因为 SQLite 的并发模式与其他数据库有所不同，对 SQLite 如何管理事务和锁，事务和锁是如何在后台工作，以及如何与您的代码一起工作等内容有所了解是很重要的。总体而言，概念不难理解，只需要在编写与 SQLite 一起工作的代码时记住几个简单规则即可。 152

接下来的章节将加深本章内容，因为这些概念不仅适用于 C API，也适用于构建在 C API 基础上的语言扩展包。

第 6 章

核心 C API

本章介绍 SQLite API 中用于处理数据库的部分。第 5 章中，我们已经了解了 API 的工作方式，现在开始关注一些具体内容。

先易后难，通过 C API 和各种有用的函数扩展示例开始我们的 SQLite 深度旅程。随着行程不断深入，您会发现 C 的等效模型一目了然，同时还有些之前可能没了解的附加功能——主要是只存在于 C API 中的一些功能。本章结束后，您应该对所有与运行命令、管理事务、获取记录、处理错误，以及与一般数据库工作相关的执行其他任务的 API 函数有整体的掌握。

SQLite v3 API 包括许多函数。不过，只有大约 8 个函数是实际处理连接、处理查询，并断开与数据库连接所必需的。其余的函数可以归入专门用于完成特定任务的小组中。

虽然直接阅读是最好的方法，但是，如果您需要关于特定函数的更详细的信息，可以查阅 C API 参考手册（网址是 www.sqlite.org/ c3ref/intro.html）。

本章所有的示例都在 examples zip 文件中，该文件可在 Apress 网站（www.apress.com）上下载。每出现一个示例时，我们会明确指出该例所对应的源文件名称。

查询封装

您已熟悉了 SQLite 执行查询的方式和各种在单个函数调用中执行 SQL 命令的封装函数。我们从 C 语言版本的 API 封装函数开始，是因为它们简单、独立，而且易于使用。这个很好的起点会让您感到有趣并且不会陷入细节中。接下来，会介绍一些与查询处理密切相关的函数。本节结束后，您可以使用封装查询对数据库进行连接、断开连接以及查询。

连接与断开连接

在执行 SQL 命令前，首先要连接数据库。也许将连接数据库称为打开数据库才是更好的表述，因为 SQLite 数据库包含在单个操作系统文件（一个文件对应一个数据库）中。同理，断开连接可以称为关闭数据库。

154

可以用 sqlite3_open_v2()、sqlite3_open()或 sqlite3_open16()函数打开数据库，这些函数的声明如下：

```
int sqlite3_open_v2(
  const char *filename,        /*数据库文件名（UTF-8）/
  sqlite3 **ppDb               /*OUT:SQLite 数据库句柄*/
  int flags,                   /*标志*/
    const char *zVfs           /*要使用的 VFS 模块的名称*/
);

int sqlite3_open (
   const void *filename,       /*数据库文件名（UTF-8）*/
   sqlite3 **ppDb              /*OUT:SQLite 数据库句柄*/
);

int sqlite3_open16(
   const void *filename,       /*数据库文件名（UTF-16）*/
   sqlite3 **ppDb              /*OUT:SQLite 数据库句柄*/
);
```

通常情况下，推荐使用 sqlite3_open_v2()，因为这是 SQLite 中最新和最强大的打开数据库的函数。与旧的 sqlite3_open()相比，它有更多的选项，具备更强的能力。不管选择哪个打开函数，参数 filename 是操作系统文件，或文本字符串':memory:'，或空字符串，或空指针。如果使用':memory:'，sqlite3_open_v2()将在内存中创建数据库，该数据库只存在于连接生存期间。如果文件名是空字符串或 null，sqlite3_open_v2()将打开临时磁盘文件，并在连接关闭时自动删除该文件。如果是其他选项，sqlite3_open_v2()将尝试打开通过参数值指定的数据库文件。如果该文件不存在，并且第三个参数包括 SQLITE_OPEN_CREATE 标志，则 sqlite3_open_v2()将通过该名称打开一个新的数据库文件，如果第三个参数不包含 SQLITE_OPEN_CREATE 标志，sqlite3_open_v2()将返回错误。

■注意：这里包含 UTF-8 和 UTF-16 版本的 sqlite3_open()声明，以及更新版本的 sqlite3_open_v2()声明。从现在起，为简洁起见，当多个版本共存时，

我们使用 UTF-8 版本的声明。我们介绍的一般是指 SQLite 最新功能的函数，而不是覆盖所有的遗留函数。因此，请牢记一点，有许多 API 函数，还有 UTF-16 形式以及用来向后兼容的旧的 UTF-8 形式。完整的 C API 文献在 SQLite 网站（www.sqlite.org/c3ref/intro.html）中可以找到。

参数 flags 包括以下比特向量：SQLITE_OPEN_READONLY、SQLITE_OPEN_READWRITE 和 SQLITE_OPEN_CREATE。这些参数都可以从名字上理解意思，只需稍加注意。SQLITE_OPEN_READONLY 和 SQLITE_OPEN_READWRITE 分别以只读模式和读/写模式打开数据库。这两个选项都要求数据库文件已经存在，否则，将返回错误。SQLITE_OPEN_CREATE 与 SQLITE_OPEN_READWRITE 结合展示前 5 章我们看到的行为，这正是传统的 sqlite3_open()行为。当数据库已经存在时，将其打开用于读取和写入操作。如果指定的数据库不存在，则会创建（尽管正如我们已经多次指出的，坚持将数据库写入磁盘会挂起第一个对象的创建）。

155

参数 flags 也可以结合 SQLITE_OPEN_NOMUTEX、SQLITE_OPEN_FULLMUTE、SQLITE_OPEN_SHAREDCACHE 或 SQLITE_OPEN_PRIVATECACHE 标志，以进一步控制数据库句柄的事务行为。最后一个参数 zVfs 允许调用去重写默认的操作系统接口 sqlite3_vfs 方法。

打开完成后，sqlite3_open_v2()将初始化由参数 ppDb 传递给它的 sqlite3 结构体。这种结构体可视为一个不透明句柄，该句柄代表到数据库的一个连接。它比数据库句柄的含义更广，因为可能将多个数据库附加到单个连接。然而，不管有多少数据库附加到该连接上，它依然代表事务上下文环境。

使用 sqlite3_close()函数可以关闭连接，该函数声明如下：

```
int sqlite3_close(sqlite3*);
```

要成功执行 sqlite3_close()，必须完成与连接关联的所有准备查询。只要有一个查询仍未完成，sqlite3_close()都会返回 SQLITE_BUSY 并显示错误消息“由于有未完成的语句，所以无法关闭连接。”

■注意：使用 sqlite3_close()关闭连接时，如果连接上有打开的事务，该事务将自动回滚。

执行查询

函数 sqlite3_exec()提供一种快速、方便执行 SQL 命令的方法，对修改数据库的命令（即，不返回任何数据的命令）特别方便。因此，通常也称为便捷函数，一个简单的 API 调用就能很好地封装很多其他任务。该函数的声明如下：

```
int sqlite3_exec(
  sqlite3*,                    /*打开的数据库*/
  const char *sql,             /*要执行的 SQL*/
  sqlite_callback,             /*回调函数*/
  void *data                   /*回调函数的第一个参数*/
  char **errmsg                /*错误信息*/
);
```

参数 sql 中提供的 SQL 可以包含多个 SQL 命令。sqlite3_exec()将解析和执行 sql 字符串中的每个命令，直到到达该字符串的末尾或遇到错误。代码 6-1（摘自 create.c）说明了如何使用 sqlite3_exec()。该示例打开一个名为 test.db 的数据库，并在该数据库中创建一个名为 episodes 的表。之后，插入一条记录。如果该数据库尚未存在，创建表命令将会从物理上创建数据库文件。

156 代码 6-1 简单命令中使用 sqlite3_exec()

```
int main(int argc, char **argv)
{
    sqlite3 *db;
    char *zErr;
    nt rc;
    char *sql;

   rc = sqlite3_open_v2("test.db", &db);

   if(rc) {
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      sqlite3_close(db);
      exit(1);
    }

    sql = "create table episodes(id int, name text)";
    rc = sqlite3_exec(db, sql, NULL, NULL, &zErr);

    if(rc != SQLITE_OK) {
```

```
        if (zErr != NULL) {
            fprintf(stderr, "SQL error: %s\n", zErr);
            sqlite3_free(zErr);
        }
    }

    sql = "insert into episodes values (10, 'The Dinner Party')";
    rc = sqlite3_exec(db, sql, NULL, NULL, &zErr);

    sqlite3_close(db);
    return 0;
}
```

正如第 5 章所述，实际上可以从 sqlite3_exec()获取记录，虽然看不到它的实现。sqlite3_exec()提供了获得 select 语句结果的回调机制，这种机制是该函数的第三和第四个参数实现的。第三个参数是指向回调函数的指针。如果提供了它，则 SQLite 将调用函数处理每个 sql 参数中执行的 select 语句的结果。回调函数具有下面的声明：

```
typedef int (*sqlite3_callback)(
    void*,    /* sqlite3_exec()函数第四个参数提供的数据*/
    int,      /*行中字段的数目*/
    char**,   /*代表行中字段名称的字符串数组*/
    char**    /*代表字段名称的字符串数组*/
);
```

sqlite3_exec()的第四个参数是 void 指针，该指针指向您想要提供给回调函数的应用程序特定的数据。SQLite 将此数据作为回调函数的第一个参数。

最后一个参数（errmsg）是指向错误消息字符串的指针，可以将处理中发生的错误消息写入该字符串。因此，sqlite3_exec()有两个错误信息来源：第一个是返回值，第二个是可读的字符串 errmsg。如果将 null 传递给 errmsg，SQLite 将不提供任何错误的消息。注意：如果提供了 errmsg 指针，用来创建错误消息的内存是在堆上分配的。因此，在调用后应该检查一下是否为 null 值，如果有错误发生，使用 sqlite3_free()释放 errmsg 所占用的内存。

157

注意：可以将 null 传递给 sqlite3_free()，其结果是执行无害的空操作。如果您想要释放，直接调用 sqlite3_free(errmsg)即可，无须检查 errmsg 是否为 null。这主要取决于您想在何处以及如何了解实际错误并采取相应的措施。

sqlite3_exec()允许执行一批命令，并可以通过回调接口收集所有的返回数据。例如，在表 episodes 中插入一条记录，然后选择所有的记录，所有的命令都在一个 sqlite3_exec() 调用中。代码 6-2 展示了完整的代码（摘自 exec.c）。

代码 6-2　使用 sqlite3_exec()处理记录

```
int callback(void* data, int ncols, char** values, char** headers);

int main(int argc, char **argv)
{
    sqlite3 *db;
    int rc;
    char *sql;
    char *zErr;

    rc = sqlite3_open_v2("test.db", &db);

    if(rc) {
       fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
       sqlite3_close(db);
       exit(1);
    }

    const char* data = "Callback function called";
    sql = "insert into episodes (id, name) values (11,'Mackinaw Peaches');"
          "select * from episodes;";
    rc = sqlite3_exec(db, sql, callback, data, &zErr);
158 if(rc != SQLITE_OK) {
        if (zErr != NULL) {
            fprintf(stderr, "SQL error: %s\n", zErr);
            sqlite3_free(zErr);
        }
    }

    sqlite3_close(db);
    return 0;
}

int callback(void* data, int ncols, char** values, char** headers)
{
    int i;
```

```
    fprintf(stderr, "%s: ", (const char*)data);
    for(i=0; i < ncols; i++) {
        fprintf(stderr, "%s=%s ", headers[i], values[i]);
    }

    fprintf(stderr, "\n");
    return 0;
}
```

SQLite 解析 sql 字符串：运行第一个命令：插入记录；然后运行第二个命令：select 语句。第二个命令中，SQLite 为每个返回的记录调用回调函数。运行该程序将产生以下输出：

```
Callback function called: id=10 name=The Dinner Party
Callback function called: id=11 name=Mackinaw Peaches
```

请注意，该回调函数将返回 0。该返回值实际上会影响 sqlite3_exec()的执行。如果回调函数返回非零值，sqlite3_exec()将终止（换句话说，它将终止当前命令和 sql 字符串中后续命令的所有处理）。

所以，sqlite3_exec()提供了一种修改数据库的简便方法，还提供了一个处理记录的接口。为什么要准备查询？下一节我们会看到使用准备查询的很多优点：

- 准备查询不需要回调接口，这样编码就更简单、清晰。
- 准备查询关联了提供列信息的函数，可以获取列的存储类型、声明类型、模式名称（如果它是别名）、表名和数据库名。sqlite3_exec()的回调接口只提供列的名称。
- 准备查询提供一种除文本外的获取字段/列值的方法，可以以 C 数据类型获取，例如 int 和 double 型，而 sqlite3_exec()的回调接口只提供字符串格式的字段值。
- 准备查询可以重新运行，可以重用已编译的 SQL。
- 准备查询支持参数化的 SQL 语句。

159

检查变化

执行更新或删除操作时，您可能想知道有多少记录受影响。可以从 sqlite3_changes()获得此信息，它提供受最后一次执行的更新、插入或删除语句影响的记录数目。很明显，如果您正在运行批处理查询（sqlite3_exec()参数中的 sql 有多个命令），该函数只适合于批处理中的最后一个命令。注意，sqlite3_changes()不包括由原始命令所激活的触发器造成的结果。

如果对具有自增长字段的表执行了插入，此时您想知道最后插入记录的主键值。这种情况下，可以使用 sqlite3_last_insert_rowid()获取此值，也可以在 SQL 内通过 SQL 函数 last_insert_rowid()获得该值。

获取表查询

函数 sqlite3_get_table()返回单独函数调用中一个命令的整个结果集。正如 sqlite3_exec()封装了准备查询 API 函数一样，可以一下运行全部的命令，sqlite3_get_table()封装了 sqlite3_exec()并尽可能便捷地返回数据。使用 sqlite3_get_table()，不必操心 sqlite3_exec()回调函数，从而使获取记录更容易。sqlite3_get_table()声明如下：

```
int sqlite3_get_table(
   sqlite3*,              /*打开的数据库*/
   const char *sql,       /*要执行的 SQL */
   char ***resultp,       /*结果写入该指针指向的 char *[ ] */
   int *nrow,             /*结果集中行的数目*/
   int *ncolumn,          /*结果集中字段的数目*/
   char **errmsg          /*错误信息*/
);
```

此函数接收 sql 中的 SQL 语句返回的所有记录，使用堆上声明的内存（使用 sqlite3_malloc()）将它们存储在参数 resultp 中。必须使用 sqlite3_free_table()释放内存，该函数将 resultp 指针作为唯一的参数。resultp 中的第一个记录实际上不是记录，而是结果集中列的名称。参考代码 6-3（摘自 get_table.c）。

160 代码 6-3 使用 sqlite3_get_table

```
int main(int argc, char **argv)
{
    /*连接数据库，等等*/

    char *result[];
    sql = "select * from episodes;";
    rc = sqlite3_get_table(db, sql, &result, &nrows, &ncols, &zErr);

    /*执行一些与数据相关的操作*/

    /*释放内存*/
```

```
    sqlite3_free_table(result)
}
```

例如，如果返回的结果集形式如下：

```
name                    | id
-------------------------
The Junior Mint         | 43
The Smelly Car          | 28
The Fusilli Jerry       | 21
```

那么，结果集数组的格式会是如下结构：

```
result [0] = "name";
result [1] = "id";
result [2] = "The Junior Mint";
result [3] = "43";
result [4] = "The Smelly Car";
result [5] = "28";
result [6] = "The Fusilli Jerry";
result [7] = "21";
```

前两个元素包含结果集中列的名称。因此，可以将结果集当成相对于行从 1 开始的索引，第 0 行对应字段名称。示例可能有助于说明这一点。代码 6-4 打印出结果中的每一行和每一列。

代码 6-4　通过 sqlite3_get_table()进行递归

```
rc = sqlite3_get_table(db, sql, &result, &nrows, &ncols, &zErr);

for(i=0; i < nrows; i++) {
    for(j=0; j < ncols; j++) {
        /*从 i+1 行开始，就跳过了第一个记录，第一个记录是存储字段名称的*/
        fprintf(stdout, "%s", result[(i+1)*ncols + j]);
    }
}
```

查询准备

161

第 5 章介绍了准备、步骤和完成函数的概念。本节会介绍该过程的全部内容，包括结果集追踪、获取记录以及使用参数化查询。可以说，准备语句比封装函数方便多了。

我们会证明了这一点。

封装函数只是将所有的步骤封装在一个函数调用中，使得某些情况下运行特定的命令更方便。每个查询函数都提供了自己的行和列的获取方式。一般而言，方法如果封装得越多，对执行和结果的控制就越少。因此，准备查询提供更多的功能、更好的控制和更多的信息，而 sqlite3_exec()提供的功能较少，sqlite3_get_table()也提供较少的功能。

准备查询使用一组特别函数访问记录的字段和列信息。列值可以使用 sqlite3_column_xxx()获得，其中，xxx 表示返回值的数据类型（例如，int、double、blob）。您可以用自己喜欢的格式检索数据，还可以获得字段的声明类型（在 create table 语句中的定义）和其他元数据，例如存储格式以及相关表和数据库名称。相比之下，sqlite3_exec()通过回调函数只提供一部分信息。sqlite3_get_table()的情况类似，该函数只提供结果集字段名称和数据。

实际应用中，您会发现每个查询函数都有其用途。sqlite3_exec()对于运行修改数据库的命令（创建、丢弃、插入、更新和删除）特别合适。一个函数调用就可以全部完成。准备查询对于 select 语句通常比较合适，因为它提供更多的信息，而且是线性编程（没有回调函数），以及通过使用游标遍历可以更好地控制结果集。

第 5 章中，我们讲到准备查询执行有三个基本步骤：编译、执行和完成。使用函数 sqlite3_prepare_v2()编译查询，使用函数 sqlite3_step()分步执行查询，使用函数 sqlite3_finalize()关闭查询，或者可以使用 sqlite3_reset()重用编译。这一过程和各步骤在下面几节中将会详细解释。

编译

编译或准备接收 SQL 语句，并将其编译为虚拟数据库引擎（VDBE）可读的字节码。这是由函数 sqlite3_prepare_v2()完成的，该函数声明如下：

```
int sqlite3_prepare_v2(
  sqlite3 *db,            /*数据库句柄*/
  const char *zSql,       /*SQL 文本，UTF-8 编码的*/
  int nBytes,             /*zSql 的字节长度*/
  sqlite3_stmt **ppStmt,  /*OUT:语句句柄*/
  const char **pzTail     /*OUT:指向 zSql 未使用的部分*/
);
```

sqlite3_prepare_v2()编译 zsql 字符串（它可以包含多条 SQL 语句）的第一个 SQL 语句。分配执行该语句所需的所有资源，并将其字节代码关联到单个语句句柄（也简称为语句），由参数 ppStmt 指定，参数 ppStmt 是 sqlite3_stmt 结构体。从程序员角度来看，这种结构体比用来执行 SQL 语句并获取相关记录的不透明句柄的内容更多。但是，这种数据结构包含了命令的字节码、绑定的参数、B-tree 游标、执行上下文以及 sqlite3_step()在执行过程中管理查询状态所需的其他数据。 162

函数 sqlite3_prepare_v2()不会对连接或数据库有任何影响，也不会启动事务或获取锁。它可以直接通过编译器工作，为执行准备查询。语句句柄高度依赖它们所被编译的数据库模式。如果另一个连接在您准备语句和实际执行语句期间更改了数据库模式，那么准备语句就会失效。但是，当模式的改变使现有语句失效时，如果可能的话，函数 sqlite3_prepare_v2()会自动试图重新编译（重准备）语句。如果模式的更改使重新编译无法实现，sqlite3_step()语句调用会导致 SQLITE_SCHEMA 错误，稍后在“错误和意外”中将讨论。此时，需要使用 sqlite3_errmsg()函数检查与 SQLITE_SCHEMA 相关的错误。

执行

一旦查询语句准备就绪，下一步就是使用 sqlite3_step()执行，sqlite3_step()声明如下：

```
int sqlite3_step(sqlite3_stmt *pStmt);
```

sqlite3_step()接收语句句柄并直接与 VDBE 通信，生成执行 SQL 语句的一个又一个步骤的字节码指令。第一次调用 sqlite3_step()时，VDBE 获得执行该命令所需的必要的数据库锁。如果不能获取锁，并且没有指派繁忙处理程序，sqlite3_step()将返回 SQLITE_BUSY。如果指派了繁忙处理程序，将调用该处理程序。

对于不返回数据的 SQL 语句，对 sqlite3_step()的第一次调用就执行完 SQL 语句，并返回一个指示结果的代码。对于返回数据的 SQL 语句，例如 select 语句，sqlite3_step()第一次调用将语句定位在第一个记录的 B-tree 光标上。后续调用的 sqlite3_step()将光标定位在结果集内的后续记录。到达末尾值之前，sqlite3_step()为结果集中的每个记录返回 SQLITE_ROW。返回 SQLITE_DONE，表示游标已到达结果集末尾。

■**注意**：对于那些熟悉旧版本 SQLite 的人而言，这些结果代码是被部分更高级的——或者更准确地说——结果代码替代了旧式的简单返回值，如

SQLITE_ERROR。结果代码以及扩展结果代码的完整列表在 www.sqlite.org/c3ref/c_abort.html 中。

与数据访问相关的所有其他 API 函数使用语句游标获取当前记录的有关信息。例如，sqlite3_column_xxx()函数都使用语句句柄，特别是游标，获取当前记录的字段。

163 完成与重置

一旦语句执行结束，必须终止。可以使用以下函数之一完成或者重置语句：

```
int sqlite3_finalize(sqlite3_stmt *pStmt);
int sqlite3_reset(sqlite3_stmt *pStmt);
```

sqlite3_finalize()将关闭语句。释放资源并提交或回滚任何隐式事务（如果该连接是自动提交模式），清除日志文件并释放相关联的锁。

如果要重复使用语句，可以使用 sqlite3_reset()。该函数将保持已编译的 SQL 语句（和任何绑定的参数），但是会将当前语句相关联的变化提交到数据库。如果启用了自动提交，它还释放锁定并清除日志文件。sqlite3_finalize()和 sqlite3_reset()之间的主要差别在于后者保留与语句关联的资源，使它可以重新执行，避免了再次调用 sqlite3_ prepare()编译 SQL 命令。

下面看一个例子。代码 6-5 显示了一个使用准备查询的简单但完整的程序，摘自 select.c 示例。

代码 6-5 使用准备查询

```
int main(int argc, char **argv)
{
    int rc, i, ncols;
    sqlite3 *db;
    sqlite3_stmt *stmt;
    char *sql;
    const char *tail;

    rc = sqlite3_open_v2("foods.db", &db);
    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
```

```
}

sql = "select * from episodes;";

rc = sqlite3_prepare_v2(db, sql, -1, &stmt, &tail);

if(rc != SQLITE_OK) {
    fprintf(stderr, "SQL error: %s\n", sqlite3_errmsg(db));
}

rc = sqlite3_step(stmt);
ncols = sqlite3_column_count(stmt);
while(rc == SQLITE_ROW) {

    for(i=0; i < ncols; i++) {
        fprintf(stderr, "'%s' ", sqlite3_column_text(stmt, i));
    }

    fprintf(stderr, "\n");

    rc = sqlite3_step(stmt);
    }

    sqlite3_finalize(stmt);
    sqlite3_close(db);

    return 0;
}
```

164

本示例连接到 foods.db 数据库、查询 episodes 表，并打印出表中记录的所有列。这只是一个简化的例子——当调用 sqlite3_step()时，我们需要检查几件事情，例如错误和繁忙条件，但是稍后我们会解决。

与 sqlite3_exec()一样，sqlite3_prepare_v2()可以接受包含多条 SQL 语句的字符串。不过，与 sqlite3_exec()不同的是，它只处理字符串中的第一条语句。但是通过 pzTail 参数，它的确使处理字符串中其余 SQL 语句变得容易。在调用 sqlite3_prepare()之后，它将该参数(如果提供了该参数)指向 zsql 字符串中的下一条语句的起始位置。使用 pzTail，可以在循环中执行给定字符串中的一批 SQL 命令，如下所示：

```
while(sqlite3_complete(sql) ){
```

```
    rc = sqlite3_prepare(db, sql, -1, &stmt, &tail);

    /*处理查询结果*/

    /*跳到字符串中的下一条命令*/
    sql = tail;
}
```

此示例使用另一个还未介绍的 API 函数——sqlite3_complete()，正如其名称那样，它采用一个字符串，如果字符串中至少有一个完整的（但不是一定有效的）SQL 语句，就返回 true，否则返回 false。实际上，sqlite_complete()寻找字符串中的终止符：分号（SQL 中的字符）。因此，虽然其名称所暗示的是某种万无一失的语句检查，实际上，它只是 sqlite 命令行中写入多行语句和做其他类似的有用任务时不断给出继续提示符的工具而已。

获取记录

到目前为止，我们已经了解了如何通过 sqlite3_exec()和 sqlite3_get_table()获取记录和列。相比之下，在从数据库获取记录信息方面，准备查询提供了更多的选择。

165 对于返回记录的语句而言，可以使用函数 sqlite3_column_count()和 sqlite3_data_count()获得结果集中的字段数，这两个函数声明如下：

```
int sqlite3_column_count(sqlite3_stmt *pStmt);
int sqlite3_data_count(sqlite3_stmt *pStmt);
```

sqlite3_column_count()返回与语句句柄关联的字段数，可以在语句实际执行之前在语句句柄上调用。如果请求不是 select 语句，则 sqlite3_column_count()返回 0。同样，sqlite3_step()返回 SQLITE_ROW 后，sqlite3_data_count()返回当前记录的列数。只有语句句柄具有活动游标时，该函数才可以工作。

获取字段信息

可以使用 sqlite3_column_name()函数获取当前记录中的所有列，该函数声明如下：

```
const char *sqlite3_column_name( sqlite3_stmt*, /*语句句柄*/
                                 int iCol       /*字段的次序*/);
```

同样，可以使用 sqlite3_column_type()函数获取每个字段关联的存储类，该函数声明如下：

```
int sqlite3_column_type( sqlite3_stmt*, /*语句句柄*/
                         int iCol       /*字段的次序*/);
```

该函数返回五个存储类对应的整数值，对应关系如下所示：

```
#define SQLITE_INTEGER    1
#define SQLITE_FLOAT      2
#define SQLITE_TEXT       3
#define SQLITE_BLOB       4
#define SQLITE_NULL       5
```

这些都是第 4 章中所述的 SQLite 原生数据类型或存储类。SQLite 数据库中的所有数据根据其最初的表示形式和列的亲和关系，以这 5 种形式之一存储。对我们而言，存储类和数据类型这两个词是同义的。有关存储类的详细信息，请参见第 4 章“存储类”和“类型关系”部分。

可以使用 sqlite3_column_decltype()函数获取字段在表模式定义中声明的数据类型，该函数声明如下：

```
const char *sqlite3_column_decltype( sqlite3_stmt*, /*语句句柄*/
                                     int            /*字段的次序*/);
```

如果在结果集中的列与实际表中的列不对应（例如，该列是文本值、表达式、函数或聚合结果），该函数将返回 NULL，作为列的声明类型。例如，假设数据库中有如下定义的表：

```
CREATE TABLE t1(c1 INTEGER);
```

166

然后，执行以下查询：

```
SELECT c1 + 1, 0 FROM t1;
```

这种情况下，sqlite3_column_decltype()第一列将返回 INTEGER，第二列为 NULL。

除了声明类型，还可以使用以下函数获得某一列的其他信息：

```
const char *sqlite3_column_database_name(sqlite3_stmt *pStmt, int iCol);
const char *sqlite3_column_table_name(sqlite3_stmt *pStmt, int iCol);
const char *sqlite3_column_origin_name(sqlite3_stmt *pStmt, int iCol);
```

第一个函数将返回与列相关联的数据库，第二个将返回它所在的表，最后一个函数将返回列在模式中的定义名称。也就是说，如果您在 SQL 语句中指定了列的别名，sqlite3_column_origin_name()将返回该类在模式定义中的实际名称。注意，这些函数只有在编译 SQLite 时启用了 SQLITE_ENABLE_COLUMN_METADATA 预处理器指令的情况下才可用。

获取字段值

可以使用 sqlite3_column_xxx()函数获取当前记录中的所有字段值，该函数的通用形式如下：

```
xxx sqlite3_column_xxx( sqlite3_stmt*, /*语句句柄*/
                        int iCol       /*字段的次序*/);
```

这里的 xxx 是您希望的数据类型（例如，int、blob、double 等）。下面是最常用的 sqlite3_column_xxx()函数：

```
int sqlite3_column_int(sqlite3_stmt*, int iCol);
double sqlite3_column_double(sqlite3_stmt*, int iCol);
long long int sqlite3_column_int64(sqlite3_stmt*, int iCol);
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);
const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
```

对于每个函数，SQLite 将内部表示形式（字段的存储类）转换成函数名字指定的类型。SQLite 有很多规则将内部数据类型表示形式转换为所请求的类型。表 6-1 列出了这些规则。

167

表 6-1　列类型转换规则

内部类型	请求类型	转　换
NULL	INTEGER	0
NULL	FLOAT	0.0
NULL	TEXT	NULL 指针
NULL	BLOB	NULL 指针
INTEGER	FLOAT	整型转换为浮点型
INTEGER	TEXT	返回该整数的 ASCII 描述
INTEGER	BLOB	返回该整数的 ASCII 描述

续表

内部类型	请求类型	转　　换
FLOAT	INTEGER	浮点型转化为整型
FLOAT	TEXT	返回该浮点数的 ASCII 描述
FLOAT	BLOB	返回该浮点数的 ASCII 描述
TEXT	INTEGER	使用 atoi()
TEXT	FLOAT	使用 atof()
TEXT	BLOB	无变化
BLOB	INTEGER	先转换成文本，然后使用 atoi()
BLOB	FLOAT	先转换成文本，然后使用 atof()
BLOB	TEXT	需要时加上\000 终结符

与稍后介绍的 sqlite3_bind_xxx()函数一样，BLOB 需要更多的工作，复制时必须指定其长度。对于 BLOB 列，可以使用 sqlite3_column_bytes()获取实际数据的长度，该函数声明如下：

```
int sqlite3_column_bytes( sqlite3_stmt*, /*语句句柄*/
                          int            /*字段的次序*/);
```

一旦获取了长度，就可以使用 sqlite3_column_blob()复制二进制数据。例如，假设在结果集中的第一列包含二进制数据，获取该数据副本的方法如下：

```
int len = sqlite3_column_bytes(stmt,0);
void* data = malloc(len);
memcpy(data, len, sqlite3_column_blob(stmt,0));
```

168

一个实例

为进一步巩固这些列函数的使用方法，代码 6-6（摘自 columns.c）说明了如何从一个简单的 select 语句使用函数检索列信息和值。

代码 6-6　获取字段信息

```
int main(int argc, char **argv)
{
    int rc, i, ncols, id, cid;
    char *name, *sql;
    sqlite3 *db;
    sqlite3_stmt *stmt;
```

```
    sql = "select id, name from episodes";
    sqlite3_open_v2("test.db", &db);

    sqlite3_prepare_v2(db, sql, strlen(sql), &stmt, NULL);

    ncols = sqlite3_column_count(stmt);
    rc = sqlite3_step(stmt);

    /*打印列信息*/
    for(i=0; i < ncols; i++) {
        fprintf(stdout, "Column: name=%s, storage class=%i, declared=%s\n",
                        sqlite3_column_name(stmt, i),
                        sqlite3_column_type(stmt, i),
                        sqlite3_column_decltype(stmt, i));
    }

   fprintf(stdout, "\n");

    while(rc == SQLITE_ROW) {
        id = sqlite3_column_int(stmt, 0);
        cid = sqlite3_column_int(stmt, 1);
        name = sqlite3_column_text(stmt, 2);
        if(name != NULL){
            fprintf(stderr, "Row: id=%i, cid=%i, name='%s'\n", id,cid,name);
        } else {
            /*字段值为 NULL */
            fprintf(stderr, "Row: id=%i, cid=%i, name=NULL\n", id,cid);
        }
        rc = sqlite3_step(stmt);
    }

169     sqlite3_finalize(stmt);
    sqlite3_close(db);
    return 0;
}
```

本示例先连接数据库，然后从 episodes 表中选择记录并打印列信息和每行（使用其内部存储类）字段。运行该程序将产生以下输出：

```
Column: name=id, storage class=1, declared=integer
Column: name=name, storage class=3, declared=text
```

```
Row: id=1, name='The Dinner Party'
Row: id=2, name='The Soup Nazi'
Row: id=3, name='The Fusilli Jerry'
```

查找语句的连接

实际应用中，您可能会发现自己编写的函数中只能访问语句句柄，而不能访问连接句柄。如果这些使用语句句柄的函数遇到错误，它们将无法从 sqlite3_errmsg()中获取错误信息，因为函数 sqlite3_errmsg()需要连接句柄。这时函数 sqlite3_db_handle()就派上用场了，该函数声明如下：

```
int sqlite3_db_handle(sqlite3_stmt*) ;
```

给定一个语句句柄，sqlite3_db_handle()会返回关联的连接句柄。通过这种方式，您就不用为在处理查询结果的时候需要到处传递关联的连接句柄而烦恼了。

查询参数化

API 支持在 SQL 语句中指定参数，允许在后面为参数提供（或“绑定”）值。绑定的参数与 sqlite3_prepare()一起使用，例如，可以创建如下所示的一个 SQL 语句：

```
insert into foo values (?,?,?);
```

然后可以进行以下操作：例如，第一个参数的整数值可以绑定为 2（指定第一个?字符），第二个参数绑定字符串值 'pi'，第三个参数绑定 double 值 3.14，代码如下所示（摘自 parameters.c）：

```
const char* sql = "insert into foo values(?,?,?)";
sqlite3_prepare(db, sql, strlen(sql), &stmt, &tail);
sqlite3_bind_int(stmt, 1, 2);
sqlite3_bind_text(stmt, 2, "pi");
sqlite3_bind_double(stmt, 3, 3.14);

sqlite3_step(stmt);
sqlite3_finalize(stmt);
```

170

上述代码产生并执行以下语句：

```
insert into foo values (2, 'pi', 3.14)
```

这种绑定方法使用位置参数（第 5 章中所述），其中参数都是用问号（?）字符指定的，然后由其索引或 SQL 语句中的相对位置识别。

深入了解其他参数方法前，最好先了解参数的定义、绑定以及评估过程。当您写了诸如以下的参数化语句时，sqlite3_prepare()编译查询时识别其中的参数：

```
insert into episodes (id,name) values (?,?)
```

sqlite3_prepare()识别到 SQL 语句中有参数。在内部，为每个参数分配一个可唯一标识的编号，位置参数从 1 开始，并以此类推使用顺序整数值。它在生成的语句句柄（sqlite3_stmt 结构体）中存储这些信息，然后期望特定值在执行前绑定到对应参数。如果参数没有绑定值，语句执行时，sqlite3_step()将默认使用 NULL 作为该参数的值。

准备语句后，可以使用 sqlite3_bind_xxx()函数绑定参数值。这些函数的一般形式如下：

```
sqlite3_bind_xxx( sqlite3_stmt*,  /* 语句句柄 */
                  int i,          /* 参数个数 */
                  xxx value       /* 绑定的值 */
                );
```

函数名称中的 xxx 代表要绑定值的数据类型。例如，将双精度值绑定到参数，可以使用 sqlite3_bind_double()，该函数声明如下：

```
sqlite3_bind_double(sqlite3_stmt* stmt, int i, double value);
```

常见的绑定函数如下：

```
int sqlite3_bind_int(sqlite3_stmt*, int, int);
int sqlite3_bind_double(sqlite3_stmt*, int, double);
int sqlite3_bind_int64(sqlite3_stmt*, int, long long int);
int sqlite3_bind_null(sqlite3_stmt*, int);
int sqlite3_bind_blob(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
int sqlite3_bind_zeroblob(sqlite3_stmt*, int, int n);
int sqlite3_bind_text(sqlite3_stmt*, int, const char*, int n, void(*)(void*));
int sqlite3_bind_text16(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
```

171

总体而言，绑定函数可分为两类，一类用于标量值（int、double、int64 和 NULL），另一类用于数组（blob、text 和 text16）。它们的区别仅在于数组绑定函数需要一个长度参数和指向清理函数的指针。此外，与 sqlite3_mprintf()函数一样，sqlite3_bind_text()自

动转义引号字符。使用 BLOB 变种类型时，数组绑定函数声明如下：

```
int sqlite3_bind_blob( sqlite3_stmt*,      /* 语句句柄*/
                       int,                /* 次序*/
                       const void*,        /* 指向 blob 数据*/
                       int n,              /* 数据的字节长度*/
                       void(*)(void*));    /* 清理处理程序*/
```

API 为清理句柄提供了如下两个有特殊含义的预定义值。

```
#define SQLITE_STATIC       ((void(*)(void *))0)
#define SQLITE_TRANSIENT    ((void(*)(void *))-1)
```

每个值指定一个特定的清理操作。SQLITE_STATIC 告诉数组绑定函数数组内存驻留在非托管的空间，所以 SQLite 不会试图清理该空间。SQLITE_TRANSIENT 告诉绑定函数数组内存经常变化，因此，SQLite 需要使用自己的数据副本，该数据副本在语句终结时会自动清除。第三个选项提供一个指向您自己的清理函数的指针，该清理函数形式必须如下：

```
void cleanup_fn(void*)
```

如果提供了自定义清理函数，语句结束时，SQLite 将调用该清理函数，并将数组内存传入。

■注意：绑定的参数在语句句柄生命周期内依然是绑定的，甚至在调用 sqlite3_reset()后，依然是绑定的。只有在该语句完成时（通过调用 sqlite3_finalize()），才被释放。

一旦绑定想要的所有参数，就可以执行该语句。可以使用序列中的下一个函数：sqlite3_step()来执行。sqlite3_step()将接收绑定值，替换 SQL 语句中的对应参数，然后开始执行命令。

现在，您已了解了绑定过程，4 个参数绑定方法的唯一不同点如下：

- 参数在 SQL 语句中的表示方式（使用位置参数，显式定义参数编号或字母数字化的参数名称）。
- 参数分配编号的方式。

对于位置参数，sqlite3_prepare()分配从 1 开始的顺序整数值。前面的示例中，第一个问号（?）分配 1，第二个问号（?）分配 2。有了位置参数，在 SQL 语句中跟踪哪个 172

号码、对应哪个参数（或问号），以及正确指定绑定函数中的编号就是您的工作。

参数编号

允许为参数指定编号，即给参数编号，而不是必须使用内部的序列。参数编号的语法是问号后紧跟一个数字。例如，下面这段代码（摘自 parameters.c）：

```
name = "Mackinaw Peaches";
sql  = "insert into episodes (id, cid, name) "
       "values (?100,?100,?101)";

rc = sqlite3_prepare(db, sql, strlen(sql), &stmt, &tail);

if(rc != SQLITE_OK) {
    fprintf(stderr, "sqlite3_prepare() : Error: %s\n", tail);
    return rc;
}

sqlite3_bind_int(stmt, 100, 10);
sqlite3_bind_text(stmt, 101, name, strlen(name), SQLITE_TRANSIENT);
sqlite3_step(stmt);
sqlite3_finalize(stmt);
```

此示例使用 100 和 101 作为参数的编号。参数编号 100 绑定整数值 10。参数 101 绑定'Mackinaw Peaches'字符串。若需要将一个值绑定到 SQL 语句中的多个位置时，要如何实现？参考如下例子：

```
insert into episodes (id, cid, name) values (?100,?100,?101)";
```

参数编号 100 使用两次——一次为 id，另一次是 cid。因此，当您需要将值绑定到多个地方时，这样可以节省时间。

■**注意**：在 SQL 语句中给参数编号时，请记住允许的范围是整数值 1～999。若要取得最优的性能和内存利用率，您应选择较小的数字。

173

参数命名

第三种参数绑定方法是给参数命名。可以在指定参数编号的地方给参数指定一个名

称。给参数编号的语法是前缀为一个问号（?），而标识命名的参数是在前面加上一个冒号（:）或@符号（@）的参数的名称。请参考下面的代码（摘自 parameters.c）：

```
name = "Mackinaw Peaches";
sql = "insert into episodes (id, cid, name) values (:cosmo,:cosmo,@newman)";

rc = sqlite3_prepare(db, sql, strlen(sql), &stmt, &tail);

sqlite3_bind_int( stmt,
                sqlite3_bind_parameter_index(stmt, ":cosmo"), 10);

sqlite3_bind_text( stmt,
                sqlite3_bind_parameter_index(stmt, "@newman"),
                name,
                strlen(name), SQLITE_TRANSIENT );

sqlite3_step(stmt);
sqlite3_finalize(stmt);
```

该示例与使用编号的参数方法相同，除非它使用的是两个命名的参数：cosmo 和 @newman 之外。与位置参数类似，sqlite3_prepare()函数也会为命名的参数自动分配编号。虽然分配给每个参数的编号是未知的，但是可以使用 sqlite3_bind_parameter_index()获得编号，该函数以参数名为输入，返回对应的参数编号，该编号也可以用来绑定参数值。总之，命名参数可以极大地提高可读性。

■**注意**：函数 sqlite3_bind_parameter_index()将参数编号称为索引，几乎所有的场合下，这两个词是同义的（编号和索引）。

Tcl 参数

最后一个参数模式称为 Tcl 参数，它是 Tcl 的特定扩展。基本上，它与参数命名完全相同，只是它使用一些变量作为参数名称，它使用 Tcl 的变量名。Tcl 扩展中调用相当于 sqlite3_prepare()的 Tcl 函数时，Tcl 扩展自动搜索与给定参数名相同的活跃的 Tcl 变量，并将它们绑定到各自的参数。尽管当前该功能只存在于 Tcl 接口中，但是并不禁止相同的机制应用到其他语言接口，其他语言也可以实现相同的功能。从这个角度看，将这种方法命名为 Tcl 参数可能有点用词不当。Tcl 扩展只是碰巧是第一个使用这种方法的应用

174 程序。Tcl 参数语法基本上只比命名参数多做一点工作——不是前面的参数加上冒号（:）或（@）符号，而是使用一个美圆符号（$）。

错误与异常

目前为止，我们从较乐观的角度看 API，是否不会出任何错，但实际中肯定会有出错情况，因此，一部分 API 用于专门处理这些情况。三种一定要在代码中关注的情况是：错误、繁忙情况以及模式更改。

错误处理

许多 API 函数返回整数结果代码。这意味着它们有可能返回某种类型的错误代码。常见的需要查看返回代码的函数有 sqlite3_open_v2()、sqlite3_prepare_v2()和 sqlite3_exec()。编程时，在使用所有的 API 函数前，应该确保处理了可能出现的错误情况。SQLite 中定义的所有错误结果只有一小部分与实际应用程序有关。表 6-2 列出了所有的 SQLite 结果代码，可以返回结果代码的 API 函数如下：

```
sqlite3_bind_xxx()
sqlite3_close()
sqlite3_create_collation()
sqlite3_collation_needed()
sqlite3_create_function()
sqlite3_prepare_v2()
sqlite3_exec()
sqlite3_finalize()
sqlite3_get_table()
sqlite3_open_v2()
sqlite3_reset()
sqlite3_step()
```

您可以使用 sqlite3_errmsg()获取给定错误的详细信息，该函数声明如下：

```
const char *sqlite3_errmsg(sqlite3*);
```

它将连接句柄作为其唯一参数，返回该连接上 API 调用产生的最近的错误。如果没有错误，它返回字符串“not an error。”

表 6-2　SQLite 结果代码

代　码	代码含义
SQLITE_OK	操作成功
SQLITE_ERROR	通常是 SQL 错误或缺少数据库。也许可以根据错误情况获取更多有关的信息（例如，SQLITE_SCHEMA）
SQLITE_INTERNAL	内部逻辑错误
SQLITE_PERM	没有访问权限
SQLITE_ABORT	回调函数请求中止
SQLITE_BUSY	数据库文件被锁
SQLITE_LOCKED	数据库中的表被锁
SQLITE_NOMEM	数据库操作中对 malloc()调用失败
SQLITE_READONLY	试图对只读数据库进行写操作
SQLITE_INTERRUPT	操作被 sqlite3_interrupt()中止
SQLITE_IOERR	某种磁盘 I/O 错误发生
SQLITE_CORRUPT	数据库磁盘文件损坏。试图将非 SQLite 数据库文件打开也会出现同样的错误
SQLITE_FULL	因为数据库已满，所以插入操作失败。文件系统没有空间或者数据库文件无法扩展
SQLITE_CANTOPEN	SQLite 无法打开数据库文件
SQLITE_PROTOCOL	数据库已锁或者协议错误
SQLITE_EMPTY	（内部使用）数据库表是空的
SQLITE_SCHEMA	数据库模式改变
SQLITE_CONSTRAINT	因为违反约束操作中止
SQLITE_MISMATCH	数据类型不匹配。例如，试图将非整型数据插入标记为 INTEGER PRIMARY KEY 的字段。SQLite 忽略大部分字段的数据类型，允许存储任何类型的数据。但是 INTEGER PRIMARY KEY 只允许存储整型数据
SQLITE_MISUSE	程序库使用不正确。当 SQLite API 程序使用不正确时可能产生该错误
SQLITE_NOLFS	使用了本机器不支持的 OS 功能。如果 SQLite 编译时启用了大文件支持（LFS）特性，但是所在的操作系统不支持 LFS，产生该错误
SQLITE_AUTH	授权被否决。使用 sqlite3_set_authorizer()安装回调函数时返回了 SQLITE_DENY
SQLITE_FORMAT	辅助性的数据库格式错误
SQLITE_RANGE	sqlite3_bind()的第二个参数超出范围
SQLITE_NOTADB	打开的文件不是 SQLite 数据库文件
SQLITE_ROW	sqlite3_step()另一行数据已经就绪
SQLITE_DONE	sqlite3_step()完整执行

虽然非嵌入式系统很少遇到 SQLITE_NOMEM 错误，但是它是嵌入式系统中遇到的最严重的错误之一，它意味着堆上没有内存可以分配（例如，malloc()失败）。SQLite 在

这方面表现得很健壮，可以优雅地从内存不足的错误中恢复。如果底层操作系统能完成像 malloc()调用的内存分配系统，SQLite 将继续工作。

繁忙情况处理

有关处理查询的两个重要函数是 sqlite3_busy_handler()和 sqlite3_busy_timeout()。如果您的程序所使用的数据库中有其他活动连接，可能会发生冲突，最终不得不等待锁，因此，将不得不处理 SQLITE_BUSY。每当调用需要获取锁的 SQLite API 函数并且 SQLite 无法得到锁时，函数就会返回 SQLITE_BUSY。处理这种情况的方法有三种：

- 自己处理 SQLITE_BUSY，通过重新运行该语句，或者采取一些其他操作。
- 让 SQLite 调用繁忙处理程序。
- 让 SQLite 等待（阻塞或睡眠）一段锁解除的时间。

最后一个选项涉及 sqlite3_busy_timeout()函数。该函数告诉 SQLite 在返回 SQLITE_BUSY 前需要等待多长的锁清除时间。虽然，最终会导致您仍不得不处理 SQLITE_BUSY，实际中，通常将此值设置为足够的时间（比如 30 秒）足以完成甚至最密集的事务。然而，在处理 SQLITE_BUSY 时仍应该有一些应急计划。

用户定义的繁忙处理

第二个选项需要使用 sqlite3_busy_handler()。该函数提供了调用用户定义函数的方法，而不是立即去阻塞或返回 SQLITE_BUSY。该函数声明如下：

```
int sqlite3_busy_handler(sqlite3*, int(*)(void*,int), void*);
```

第二个参数是指向繁忙处理程序的函数指针，第三个参数是指向应用程序特定数据的指针，它将作为繁忙处理函数的第一个参数。繁忙处理程序的第二个参数是事先向同一个锁事件的处理程序的调用次数。

繁忙处理程序可能调用 sleep()等待一段时间，等待锁，或者它可能发出某种形式的通知。它能执行任何您想让它执行的事情，因为该函数是由您实现的。注意，注册一个繁忙处理程序并不能保证它将总是被调用。正如第 5 章所述，如果 SQLite 检测到可能产生死锁，那么它将放弃为连接调用繁忙处理程序。具体地说，如果您的 SHARED 连接干扰了另一个 RESERVED 连接，SQLite 将不会调用繁忙处理程序，这种情况下，您应该从 RESERVED 状态（BEGIN IMMEDIATE 开始的事务会从 RESERVED 状态执行

数据库写操作）执行数据库写操作，而您实际上却试图从 SHARED 状态（从 BEGIN 开始的事务会从 SHARED 状态执行数据库写操作）执行数据库写操作。

繁忙处理程序的唯一局限是它们可能不会关闭数据库。在繁忙处理程序中关闭数据库，会删除繁忙处理程序外面的执行查询的关键数据结构，并导致程序崩溃。

建议

考虑到各种情况，如果您收到了 SQLITE_BUSY 值，最佳方法可能是设置合理的超时时间，然后采取一些预防措施。一般情况下，如果您要写数据库，从 RESERVED 开始。如果您不写数据库，退而求其次，安装一个繁忙处理程序，将超时设置为一个已知值，如果 SQLite 返回 SQLITE_BUSY，检查响应时间。如果时间小于繁忙处理程序的延迟，然后 SQLite 告诉您查询（和连接）阻止写操作。如果此时想写数据库，应完成或重置，从 BEGIN IMMEDIATE 开始，然后重新执行该语句。

模式改变处理

当连接更改了数据库模式，所有在改变发生前编译的准备语句都将失效。其结果是，这种语句的第一个 sqlite3_step()调用将尝试重新编译有关 SQL 并尽可能正常进行。如果 178 重新编译时无法实现（例如，如果某个对象已被完全删除），则函数 sqlite3_step()返回错误代码 SQLITE_SCHEMA。从锁定的角度来看，模式更改发生的时间介于对编译语句 sqlite3_prepare()的调用和执行语句 sqlite3_step()的调用之间。

这种情况发生时，唯一的应对方法就是处理改变并重新开始。以下一些事件可能会导致 SQLITE_SCHEMA 错误：

- 分离数据库。
- 修改或安装用户自定义的函数或聚合。
- 修改或安装用户自定义的排序规则。
- 修改或安装授权函数。
- 清理数据库空间。

SQLITE_SCHEMA 情况存在的最终原因与 VDBE 有关。当连接更改模式时，其他已编译查询的 VDBE 代码可能会指向一个已经不存在，或者位置已经改变的数据库对象。为了避免运行时产生奇怪的错误，SQLite 将已编译但未执行的所有语句无效化，它

们必须重新编译。

追踪 SQL

如果您很难弄清楚程序在数据库中到底做了什么操作，可以使用 sqlite3_trace()跟踪 SQL 语句执行的操作。sqlite3_trace()声明如下：

```
void *sqlite3_trace(sqlite3*, void(*xTrace)(void*,const char*), void*);
```

该函数类似于在连接上放窃听器。您可以使用它来生成给定连接上执行的所有 SQL 的日志文件以及提供有用的调试信息。处理的每个 SQL 语句都将传递给第二个参数中指定的回调函数。SQLite 将 sqlite3_trace()第三个参数提供的数据传递给回调函数作为其第一个参数。

操作控制

API 提供了一些函数，可以在编译时和运行时监视和/或管理 SQL 命令。这些函数可以安装用来监视数据库的回调函数，通常在各种数据库事件发生时控制这些事件。

提交钩子

函数 sqlite3_commit_hook()可以监视给定连接上的事务提交事件。该函数声明如下：

179
```
void *sqlite3_commit_hook( sqlite3 *cnx,                    /*数据库句柄*/
                           int(*xCallback)(void *data), /*回调函数*/
                           void *data);                     /*应用程序数据*/
```

该函数注册回调函数 xCallback，当连接 cnx 上提交事务时将触发该回调函数。第三个参数（data）是一个指向应用程序特定数据的指针，SQLite 将该数据传递给回调函数。如果回调函数返回非零值，提交将转换为回滚。

在回调函数指针中传入 NULL，可以有效地禁用当前注册函数。此外，给定的连接一次只能注册一个回调函数。如果之前没有注册过，sqlite3_commit_hook()的返回值为 NULL。如果以前注册过另一个回调函数，sqlite3_commit_hook()将返回以前的参数 data 值。

回滚钩子

回滚钩子与提交钩子类似，不同之处在于回滚钩子关注给定连接上的回滚事件。回滚钩子使用如下函数注册：

```
void *sqlite3_rollback_hook(sqlite3 *cnx, void(*xCallback)(void *data), void
*data);
```

该函数注册回调函数 xCallback，连接 cnx 上的回滚事件触发回调函数，不管是通过显式的回滚命令、隐式错误还是违反约束导致的回滚。如果是由于数据库连接关闭导致的自动回滚，将无法触发回调函数。第三个参数（data）是指向应用程序特定的数据指针，SQLite 将其传递给回调函数。

与 sqlite3_commit_hook()一样，每次调用 sqlite3_rollback_hook()时，提供的新回调函数将替换以前注册的回调函数。如果之前已经注册过回调函数，则 sqlite3_rollback_hook()返回原来的数据参数。

更新钩子

函数 sqlite3_update_hook()用于监视给定连接上对行的所有更新、插入和删除操作。该函数形式如下：

```
void *sqlite3_update_hook(
  sqlite3 *cnx,
  void(*)(void *, int, char const*, char const*, sqlite_int64),
  void *data);
```

回调函数的第一个参数是指向特定应用程序数据的指针，该参数是在函数 sqlite3_update_hook()的第三个参数中提供的。回调函数具有以下形式：

```
void callback (  void * data,
                 int operation_code,
                 char const *db_name,
                 char const *table_name,
                 sqlite_int64 rowid);
```

参数 operation_code 与代表插入、更新和删除操作的 SQLITE_INSERT、SQLITE_UPDATE 和 SQLITE_DELETE 操作对应。第三和第四个参数分别对应数据库名称和操作发生的表名称。最后一个参数是受影响行的 rowid。系统表上的操作不会调用回调函

180

数（例如，sqlite_master 表和 sqlite_sequence 表）。如果之前存在回调函数，则返回值是指向以前注册的回调函数数据的指针。

授权函数

也许最强大的事件过滤器是 sqlite3_set_authorizer()。它允许您监控或控制查询语句的编译，该函数声明如下：

```
int sqlite3_set_authorizer(
  sqlite3*,
  int (*xAuth)( void*,int,
                const char*, const char*,
                const char*,const char*),
  void *pUserData
);
```

该例程注册作为授权函数的回调函数。SQLite 在语句编译时（不是在执行时）为各种数据库事件调用回调函数。该函数的目的是使应用程序可以安全地执行用户提供的 SQL，它提供了一种限制某种 SQL 操作或否决对数据库中特定表或字段访问的方法。

为清楚起见，授权回调函数的形式如下：

```
int auth( void*,          /*用户数据*/
          int,            /*事件代码*/
          const char*,    /*事件具体相关的参数*/
          const char*,    /*事件具体相关的参数*/
          const char*,    /*数据库名称*/
          const char*     /*触发器或视图名称*/ );
```

第一个参数是特定应用程序的数据指针，在 sqlite3_set_authorizer()的第四个参数中传递。授权函数的第二个参数是表 6-3 中定义的某个常量，这些值表示授予什么类型的操作权限。授权函数的第三和第四个参数是具体的事件代码，这些参数和与之相对应的事件代码在表 6-2 中已经给出。

第五个参数是数据库（主数据库、临时数据库等）名称。第六个参数是负责访问尝试的最内层的触发器或视图，如果该值为 null，表示这种访问是直接来自顶级的 SQL 语句。

授权函数的返回值应该是常量 SQLITE_OK、SQLITE_DENY 或 SQLITE_IGNORE

中的某个。其中，前两个值对所有事件的含义一致——允许或拒绝的 SQL 语句。SQLITE_DENY 将中止整个 SQL 语句，并产生错误。

SQLITE_IGNORE 的意思与具体的事件相关。读取或修改记录的语句为它所操作的每个字段生成 SQLITE_READ 或 SQLITE_UPDATE 事件。这种情况下，如果回调函数返回 SQLITE_IGNORE，将从操作中排除问题列。具体地说，试图读取此列的数据只会得到 NULL 值，更新此列将以默认方式失败。

表 6-3　SQLite 授权事件

181

事件代码	参数 3	参数 4
SQLITE_CREATE_INDEX	Index name	Table name
SQLITE_CREATE_TABLE	Table name	NULL
SQLITE_CREATE_TEMP_INDEX	Index name	Table name
SQLITE_CREATE_TEMP_TABLE	Table name	NULL
SQLITE_CREATE_TEMP_TRIGGER	Trigger name	Table name
SQLITE_CREATE_TEMP_VIEW	View name	NULL
SQLITE_CREATE_TRIGGER	Trigger name	Table name
SQLITE_CREATE_VIEW	View name	NULL
SQLITE_DELETE	Table name	NULL
SQLITE_DROP_INDEX	Index name	Table name
SQLITE_DROP_TABLE	Table name	NULL
SQLITE_DROP_TEMP_INDEX	Index name	Table name
SQLITE_DROP_TEMP_TABLE	Table name	NULL
SQLITE_DROP_TEMP_TRIGGER	Trigger name	Table name
SQLITE_DROP_TEMP_VIEW	View name	NULL
SQLITE_DROP_TRIGGER	Trigger name	Table name
SQLITE_DROP_VIEW	View name	NULL
SQLITE_INSERT	Table name	NULL
SQLITE_PRAGMA	Pragma name	First argument or NULL
SQLITE_READ	Table name	Column name
SQLITE_SELECT	NULL	NULL
SQLITE_TRANSACTION	NULL	NULL
SQLITE_UPDATE	Table name	Column name
SQLITE_ATTACH	File name	NULL
SQLITE_DETACH	Database name	NULL

182

为了说明这一点，下面的示例（完整代码在 authorizer.c 文件）将创建表 foo，具体如下：

```
create table foo(x int, y int, z int)
```

注册授权函数，该授权函数可以完成如下操作：

- 阻止读取列 z。
- 阻止更新列 x。
- 监控加载和分离数据库事件。
- 记录数据库事件日志。

这是一个相当长的例子，使用授权函数过滤了许多不同的数据库事件。因此，为了清晰起见，我们将代码分成几部分，授权函数的一般形式如代码 6-7 所示。

代码 6-7　授权函数示例

```
int auth( void* x, int type,
          const char* a, const char* b,
          const char* c, const char* d )
{
    const char* operation = a;

    printf( " %s ", event_description(type));

    /*过滤不同的数据库事件，
    **从数据库事务事件（SQLITE_TRANSACTION）到插入事件（SQLITE_INSERT）、
    **更新（UPDATE）、删除（DELETE）、附着（ATTACH）等。
    **允许或者否决事件执行。
    */

    return SQLITE_OK;
}
```

183 授权函数的第一件事就是监控事务状态的变化。如果发现变化，它将打印一条消息：

```
if((a != NULL) && (type == SQLITE_TRANSACTION)) {
    printf(": %s Transaction", operation);
}
```

下一步是过滤导致模式改变的事件：

```
switch(type) {
```

```
case SQLITE_CREATE_INDEX:
case SQLITE_CREATE_TABLE:
case SQLITE_CREATE_TRIGGER:
case SQLITE_CREATE_VIEW:
case SQLITE_DROP_INDEX:
case SQLITE_DROP_TABLE:
case SQLITE_DROP_TRIGGER:
case SQLITE_DROP_VIEW:
{
    //以某种方式修改模式
    printf(": Schema modified");
 }
```

下一个过滤器监控任何尝试进行读取的操作（这是列基础上的）。这里，如果读取列存在并且不是列 z，允许所有的读取尝试，函数返回 SQLITE_IGNORE。SQLite 对列 z 的返回值为 NULL，这样就能有效地阻止对列 z 的访问。

```
if(type == SQLITE_READ) {
        printf(": Read of %s.%s ", a, b);

        /*阻止对字段 z 的读取操作*/
        if(strcmp(b,"z") == 0) {
            printf("-> DENIED\n");
            return SQLITE_IGNORE;
        }
}
```

下面过滤 insert 和 update 事件。允许所有的 insert 语句。但是，试图更新列 x 将被否决。这样不会阻止 update 语句的执行，只是将列 x 的更新试图排除。

```
if(type == SQLITE_INSERT) {
    printf(": Insert records into %s ", a);
}

if(type == SQLITE_UPDATE) {
    printf(": Update of %s.%s ", a, b);
    /*阻止对字段 x 的更新操作*/
    if(strcmp(b,"x")==0) {
        printf("-> DENIED\n");
        return SQLITE_IGNORE;
    }
}
```

184

最后，授权函数过滤 delete、attach 以及 detach 语句，遇到这些事件时，它只是简单地发出通知：

```
if(type == SQLITE_DELETE) {
    printf(": Delete from %s ", a);
  }

if(type == SQLITE_ATTACH) {
    printf(": %s", a);
  }

if(type == SQLITE_DETACH) {
    printf("-> %s", a);
  }

   printf("\n");
   return SQLITE_OK;
}
```

（删减版的）程序实现如下所示。与授权函数一样，我们将它分成几部分，程序的第 1 步是连接数据库并注册授权函数：

```
int main(int argc, char **argv)
{
    sqlite3 *db, *db2;
    char *zErr, *sql;
    int rc;

    /**建立*/

    /*连接数据库 test.db */
    rc = sqlite3_open_v2("test.db", &db);

    /**授权与测试

    /* 1. 注册授权函数*/
    sqlite3_set_authorizer(db, auth, NULL);
```

185

第 2 步说明事务过滤器：

```
    /* 2. 测试事务相关事件*/
```

```
printf("program : Starting transaction\n");
sqlite3_exec(db, "BEGIN", NULL, NULL, &zErr);

printf("program : Committing transaction\n");
sqlite3_exec(db, "COMMIT", NULL, NULL, &zErr);
```

第 3 步通过创建表 foo 测试模式改变通知：

```
/* 3. 测试表相关事件*/

printf("program : Creating table\n");
sqlite3_exec(db, "create table foo(x int, y int, z int)", NULL, NULL, &zErr);
```

第 4 步测试读取（select）和写入（insert/update）控制。插入测试记录，选择该记录，更新该记录，并再次选择该记录以验证更新结果。

```
/* 4. 测试读/写访问*/
printf("program : Inserting record\n");
sqlite3_exec(db, "insert into foo values (1,2,3)", NULL, NULL, &zErr);

printf("program : Selecting record (value for z should be NULL)\n");
print_sql_result(db, "select * from foo");

printf("program : Updating record (update of x should be denied)\n");
sqlite3_exec(db, "update foo set x=4, y=5, z=6", NULL, NULL, &zErr);

printf("program : Selecting record (notice x was not updated)\n");
print_sql_result (db, "select * from foo");

printf("program : Deleting record\n");
sqlite3_exec(db, "delete from foo", NULL, NULL, &zErr);

printf("program : Dropping table\n");
sqlite3_exec(db, "drop table foo", NULL, NULL, &zErr);
```

上述代码发生了几件事。程序选择表中的所有记录，其中一个字段是 z 。我们应该看到输出结果中列 z 的值为 NULL，所有其他的字段包含了表中的数据。接下来，该程序将试图更新所有的字段，其中最重要的是列 x。更新操作成功了，但列 x 中的值应维持不变，因为授权函数否决对列 x 的更新操作。这一点在下面的 select 语句中得到验证，该 select 语句显示除列 x 外，所有的列被更新了。然后该程序删除表 foo，这样触发前

面过滤器中的模式改变通知。

第 5 步测试数据库命令 attach 和 detach。需要注意的是，数据库的授权函数是如何提供名称的，以及可以区分相对于主数据库的附加数据库操作。

186
```
/* 5. 测试附着和分离*/

/* 连接数据库 test2.db */
rc = sqlite3_open_v2("test2.db", &db2);

if(rc) {
    fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db2));
    sqlite3_close(db2);
    exit(1);
}

sqlite3_exec(db2, "drop table foo2", NULL, NULL, &zErr);
sqlite3_exec(db2, "create table foo2(x int, y int, z int)",
                   NULL, NULL, &zErr);

printf("program : Attaching database test2.db\n");
sqlite3_exec(db, "attach 'test2.db' as test2", NULL, NULL, &zErr);

printf("program : Selecting record from attached database test2.db\n");
sqlite3_exec(db, "select * from foo2", NULL, NULL, &zErr);

printf("program : Detaching table\n");
sqlite3_exec(db, "detach test2", NULL, NULL, &zErr);
```

再次为清楚起见，这里将程序分成几个片段。执行后，我们看到的第一件事是事务过滤器捕获事务状态的变化：

```
program : Starting transaction
    SQLITE_TRANSACTION : BEGIN Transaction
program : Committing transaction
    SQLITE_TRANSACTION : COMMIT Transaction
program : Starting transaction
    SQLITE_TRANSACTION : BEGIN Transaction
program : Aborting transaction
    SQLITE_TRANSACTION : ROLLBACK Transaction
```

接下来，我们看到创建测试表 foo 导致模式改变。注意，有趣的事情发生了，创建

测试表引起的 sqlite_master 表改变事件出现了：

```
program : Creating table
    SQLITE_INSERT : Insert records into sqlite_master
    SQLITE_CREATE_TABLE : Schema modified
    SQLITE_READ : Read of sqlite_master.name
    SQLITE_READ : Read of sqlite_master.rootpage
    SQLITE_READ : Read of sqlite_master.sql
    SQLITE_UPDATE : Update of sqlite_master.type
    SQLITE_UPDATE : Update of sqlite_master.name
    SQLITE_UPDATE : Update of sqlite_master.tbl_name
    SQLITE_UPDATE : Update of sqlite_master.rootpage
    SQLITE_UPDATE : Update of sqlite_master.sql
    SQLITE_READ : Read of sqlite_master.ROWID
    SQLITE_READ : Read of sqlite_master.name
    SQLITE_READ : Read of sqlite_master.rootpage
    SQLITE_READ : Read of sqlite_master.sql
    SQLITE_READ : Read of sqlite_master.tbl_name
```

187

下一步程序是插入一条记录，授权函数检测到该变化：

```
program : Inserting record
    SQLITE_INSERT : Insert records into foo
```

这里的事情变得更有趣。我们将看到授权阻止对单个列的访问，程序选择表 foo 中的所有记录。我们看到 SQLITE_SELECT 事件的发生，随后是 select 语句对每一列的尝试访问生成的 SQLITE_READ 事件。当到达列 z 时，授权函数拒绝访问。之后，SQLite 执行语句，print_sql_result()打印结果集中的字段信息和行：

```
program : Selecting record (value for z should be NULL)
    SQLITE_SELECT
    SQLITE_READ : Read of foo.x
    SQLITE_READ : Read of foo.y
    SQLITE_READ : Read of foo.z -> DENIED
  Column: x (1/int)
  Column: y (1/int)
  Column: z (5/(null))
  Record: '1' '2' '(null)'
```

看看列 z 发生了什么。它的值为 NULL，这一点验证了授权函数对该列访问的否决。

查看列信息可知，虽然 SQLite 显示了列 z 的存储类，但是它拒绝对其在模式中声明类型的访问。

接下来是更新操作。这里，我们感兴趣的是列 x。update 语句试图更改所有记录中的值，但是对列 x 的更新被拒绝：

```
program : Updating record (update of x should be denied)
    SQLITE_UPDATE : Update of foo.x -> DENIED
    SQLITE_UPDATE : Update of foo.y
    SQLITE_UPDATE : Update of foo.z
```

要确认这一点，可以执行查询操作查看发生了什么。Update 操作的确执行了，但列 x 未更改。更有趣的是，虽然 z 已更新（这是事实），但授权函数让我们看不到改变：

```
188 program : Selecting record (notice x was not updated)
    SQLITE_SELECT
    SQLITE_READ : Read of foo.x
    SQLITE_READ : Read of foo.y
    SQLITE_READ : Read of foo.z -> DENIED
  Column: x (1/int)
  Column: y (1/int)
  Column: z (5/(null))
  Record: '1' '5' '(null)'
```

下一步程序是删除该记录，并删除表。如同创建表一样，删除表在 sqlite_master 表中生成各种各样的事件：

```
program : Deleting record
    SQLITE_DELETE : Delete from foo
program : Dropping table
    SQLITE_DELETE : Delete from sqlite_master
    SQLITE_DROP_TABLE : Schema modified
    SQLITE_DELETE : Delete from foo
    SQLITE_DELETE : Delete from sqlite_master
    SQLITE_READ : Read of sqlite_master.tbl_name
    SQLITE_READ : Read of sqlite_master.type
    SQLITE_UPDATE : Update of sqlite_master.rootpage
    SQLITE_READ : Read of sqlite_master.rootpage
```

最后，程序在一个单独连接上创建另一个数据库，然后将其附加到主连接上。主连接再从附加数据库的表中选择记录，我们可以看到授权函数如何报告发生在附加数据库

中的这些操作：

```
program : Attaching database test2.db
    SQLITE_ATTACH : test2.db
    SQLITE_READ : Read of sqlite_master.name
    SQLITE_READ : Read of sqlite_master.rootpage
    SQLITE_READ : Read of sqlite_master.sql
program : Selecting record from attached database test2.db
    SQLITE_SELECT
    SQLITE_READ : Read of foo2.x
    SQLITE_READ : Read of foo2.y
    SQLITE_READ : Read of foo2.z -> DENIED
program : Detaching table
    SQLITE_DETACH -> test2
```

正如您看到的，函数 sqlite3_set_authorizer()与其事件过滤功能相当强大。它为您提供了在给定数据库上用户能做什么和不能做什么的强大的控制手段。您可以监视事件，如果您愿意，可以在事件实际执行前阻止它；或者，可以允许它们继续执行，但是对它们施加一些具体的限制。sqlite3_set_authorizer()是处理 SQLITE_SCHEMA 情况的很有用的工具。如果您的应用程序数据库模式不需要改变，可以直接安装一个授权函数，拒绝任何以某种方式试图修改模式的操作。

> ■注意：请记住，在授权函数中否决对模式的改变并不是 SQLITE_SCHEMA 事件的灵丹妙药。您需要仔细关注否定各种模式改变所造成的影响，一味阻止所有导致模式改变的事件会限制其他貌似合法的操作，如 VACUUM。

交互式程序的帮助

SQLite 提供了两个使得与交互程序一起工作变得更容易的函数。第一个是函数 sqlite3_interrupt()，声明如下：

```
void sqlite3_interrupt(sqlite3* /* connection handle */);
```

sqlite3_interrupt()使给定连接上的悬挂数据库操作终止，并尽早返回。该程序是在响应用户中断行为时调用的，例如，用户单击图形应用程序中的取消按钮或在命令行程序中按组合键 Ctrl + C 调用此例程。它旨在适应这种情况，用户希望立即停止运行时间长的查询操作。命令行程序中，可以将此函数放到信号处理程序或图形应用程序的事件处理程序中。

用于交互程序的另一个函数是sqlite3_progress_handler()，它在功能上与函数sqlite3_interrupt()有些关联，但也有些不同。该函数声明如下：

```
void sqlite3_progress_handler( sqlite3*,      /*连接句柄*/
                               int frq,       /*回调的频率*/
                               int(*)(void*), /*回调函数*/
                               void*);        /*应用程序数据*/
```

该函数安装一个会在sqlite3_exec()、sqlite3_step()和sqlite3_get_table()调用期间被触发的回调函数。该函数的目的是让应用程序在长时间运行的查询过程中为用户提供反馈信息。

第二个参数（frq）以VDBE指令形式指定回调频率。也就是说，查询执行中每个frq VDBE指令都会调用进度回调函数（第三个参数提供的）。如果对sqlite3_exec()、sqlite3_step()或sqlite3_get_table()的调用要求小于要执行的frq指令，那么进度回调函数将不被调用。第四个参数是指向特定应用程序数据的指针，该指针传递给进度处理程序（作为它的一个并且是仅有的一个参数）。如果回调函数赋值为NULL，则会禁用当前已安装的进度处理程序（如果有安装过）。

如果进度回调函数返回非零值，将立即终止当前查询，数据库中的任何更改将被回滚。如果查询是较大事务的一部分，那么事务不会回滚，继续保持活动状态。在这种情况下，sqlite3_exec()调用将返回SQLITE_ABORT。

线程

SQLite的很多版本都支持多线程。虽然多线程代码中常见的代码保护技术都适用于SQLite，但还是可以采取一些额外的预防措施。

与线程相关的一个限制是UNIX平台上的系统调用fork()。决不要尝试在fork()调用中将连接传递给子进程，否则将无法正常工作。

共享缓存模型

从SQLite 3.3.0开始，SQLite支持共享缓存模式，如图6-1所示，允许一个进程中的多个连接使用共同的页缓存。该功能用于单个线程能够代表其他线程有效管理多个数

据库连接的嵌入式服务器。这种情况下，连接共享单个页缓存以及不同的并发模型。与每个线程管理自己的连接相比较，由于共享缓存，这种服务器线程运行时可以大大降低内存使用，并具有更好的并发性。通常情况下，每个连接需要分配足以容纳 2000 个页面的内存作为页缓存。共享缓存模式与其他线程共享一个页缓存，不用每个线程都消耗这么多内存。

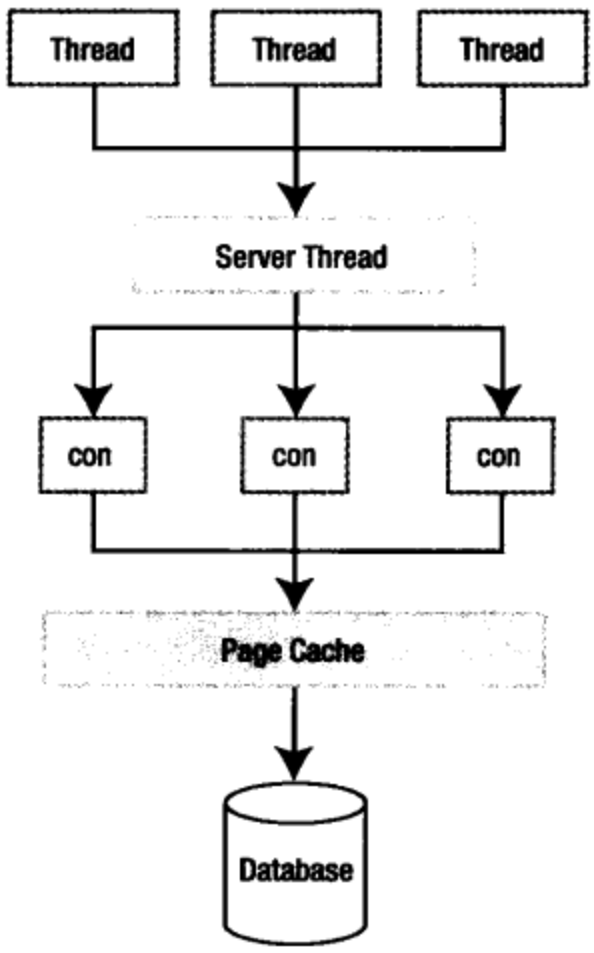


图 6-1　共享缓存模式

共享缓存模式中，线程依赖于服务器线程帮它们管理数据库连接。线程通过某种通信机制向服务器发送 SQL 语句；服务器使用线程分配的连接执行它们，然后返回结果。线程可以继续发出命令并控制自己的事务，只是实际连接存在于其他线程并由其他线程管理。

共享缓存模式中的连接使用不同的并发模型和隔离级别。此外，有个选项可以允许共享缓存中的连接看到它们同族连接的更改内容。如果编译指示 read_committed 设置为 true，这种改变检查是可能的。也就是说，在数据库中可以同时有多个读操作和一个活跃的写操作。那个写操作可以写入数据库——执行完整的写事务——而无须等待清除其

他的读操作。因此，处于读事务中的数据库数据可以改变。虽然 SQLite 通常在一个序列化的隔离级别运行，也就是说，读操作和写操作总是看到一致的数据库，在共享缓存模式中，读操作有可能看不到数据库背后发生的变化。

有些控制措施可以让连接不会阻挡他人。默认情况下，共享缓存模式使用表锁分别保持读连接和写连接。表锁与数据库锁不同，它只在共享缓存下的连接中存在。当连接读取表时，SQLite 首先试图获取表上的读锁定。写表也同样如此。连接不可以向另一个已经有读锁定连接的表写入，反之亦然。此外，表锁绑定到各自的连接并且只在其事务内持续有效。

读未提交隔离级别

虽然表锁有助于将连接保持在共享缓存模式下，不影响他人使用。也就是说，通过使用编译指示 read_uncommitted，连接可以选择读未提交隔离级别。如果将 read_uncommitted 设为 true，则该连接不会在读取的表上加读锁。因此，另一个写操作实际上可以在连接读取的时候更改表。这可能导致不一致的查询结果，但它也意味着读取未提交模式中的连接既不阻止其他连接，也不会被其他连接阻止。

192

SQLite 3.7 采用了预写日志模式，因此，发生了一些大的变化。我们将在第 11 章详细介绍预写日志记录。

共享缓存模型中的模式改变

当线程想要修改数据库模式时，共享缓存模式还有一些需要考虑的事情。事实证明，SQLite 发起表锁之前，首先在 sqlite_master 表上获取表级读取锁。修改模式的线程，必须先获得 sqlite_master 上的表级写锁。只要没有来自其他表的读取锁，就可以获得 sqlite_master 上的表级写锁。当一个线程获得 sqlite_master 写锁时，在此期间，不允许其他线程编译 SQL 语句。否则，其他线程会得到 SQLITE_SCHEMA 错误。

解锁通知

如果您对读未提交隔离这种操作思想不感兴趣，SQLite 提供了一种依然使用共享缓存模式的另一种方法，最新版本的 SQLite 包含 sqlite3_unlock_notify()函数，在锁定情况下提供了额外的灵活性，sqlite3_unlock_notify()函数声明如下：

```
int sqlite3_unlock_notify(
  sqlite3 *pBlocked,                          /*等待的连接*/
```

```
    void (*xNotify)(void **apArg, int nArg),     /*要触发的回调函数*/
    void *pNotifyArg                             /*传递给 xNotify 的参数*/
)
```

如果您的代码因为存在锁竞争未能获得共享锁而失败，将返回 SQLITE_LOCKED。这种情况下，可以调用 sqlite3_unlock_notify()，在阻塞连接（第一个参数）上注册回调函数 xNotify（第二个参数），回调函数可以有任何参数（通过第三个参数 pNotifyArg 指针访问）。

拥有阻塞锁的连接将会触发 xNotify 回调，作为完成事务的 sqlite3_step()或 sqlite3_close()调用的一部分。特别是在多线程应用程序中调用 sqlite3_unlock_notify()时，正在完成的事务就完成了，这是有可能的。在这种情况下，回调函数将立即开始从 sqlite3_unlock_notify()内初始化。

使用 sqlite3_unlock_notify()有以下注意事项：

- 每个被阻塞连接上只能注册一个解锁/通知回调函数（较新的回调函数会替换现有的回调）。
- sqlite3_unlock_notify()不是可重入的，这意味着您不应在回调函数中包括其他 SQLite 函数调用。否则，可能会使应用程序崩溃。
- 如果使用 drop table 或者 drop index 命令，就需要特别注意。因为这些语句会被 select 语句“阻塞”，参数 pBlocked 中没有包含真正阻塞的锁事务。要使用扩展返回代码检测并处理这种情况，可参考 SQLite 网站（www.sqlite.org/c3ref/unlock_notify.html）中的详细信息。 193

线程与内存管理

因为共享缓存模式与节省内存相关，SQLite 有几个与线程和内存管理相关联的函数，它们可以指定堆大小、弹性堆栈限制，以及手动启动内存清理。这些函数如下：

```
void sqlite3_soft_heap_limit(int N);
int sqlite3_release_memory(int N);
```

函数 sqlite3_soft_heap_limit()将调用线程的当前软堆设置为 N 字节。如果使用中线程的堆超过 N，那么 SQLite 将自动调用 sqlite3_release_memory()，试图从与该调用线程相关的所有数据库连接中释放至少 N 字节的内存。sqlite3_release_memory()的返回值是

实际释放的字节数。

这些例程在默认情况下都是空操作，除非在编译 SQLite 时启用预处理器指令 SQLITE_ENABLE_MEMORY_MANAGEMENT 以启用内存管理。

总结

核心 C API 包含处理 SQL 命令所需要的一切，它包含多种便捷的查询方法，在不同的场合下很有用。这些方法包括 sqlite3_exec()和 sqlite3_get_table()，使您可以在单个函数调用中执行命令。API 包括许多实用功能，这些功能有助于您知道查询中受影响的记录数，获取 INSERT 语句最后一个插入的 ROWID，追踪运行在连接中的 SQL 命令以及格式化 SQL 语句字符串。

函数 sqlite3_prepare_v2()、sqlite3_step()、sqlite3_finalize()为您提供了很大的灵活性，同时，通过使用语句句柄，它们可以控制语句的执行。语句句柄提供更详细的行和列信息，方便绑定参数以及能够重用预准备语句，避免查询编译开销。

通过使用事件过滤器，SQLite 提供多种方法管理运行时事件。提交、回滚和更新钩子允许您监视和控制数据库中特定状态的变化。在编译时使用授权函数可以监视和限制查询能做什么，这样可以观察到行和列的变化。

不管您是否相信，目前您只看到了 SQLite API 的一半内容！实际上，差不多是四分之三的内容，但您很可能会发现，下一章内容——用户自定义函数、聚合和排序规则——每一项都有用并且有趣。

第 7 章

扩展 C API

本章介绍有关 SQLite 的新技术。第 6 章介绍数据库的一般操作，本章将介绍一些创造性的内容。扩展 API 提供几种扩展或定制 SQLite 的基本方法，包括：创建用户自定义函数、聚合、排序规则和虚拟表。还可以在更底层实现一些方法，像虚拟文件系统，启动时可更换的页缓存、内存分配和互斥的实现。本章主要讨论前三个部分：用户自定义函数、聚合和排序规则。

用户自定义函数是用户编写的用于特定应用的 SQL 函数，可以在 SQL 语句中调用它们。例如，可以创建一个返回字符串'Hello Jerry'的函数 hello_newman()，一旦注册该函数，可以按照如下方式调用：

```
sqlite > select hello_newman() as reply;
reply
------------------
'Hello Jerry'
```

这就是定制的包含函数 hello_newman()的特殊版本的 SQLite。

聚合是一种特殊形式的函数，它与普通函数的工作方式基本相同，只是它对一组记录进行操作并返回这一组记录中某个特定字段的聚合值或表达式计算值，也就是说，聚合函数可以从多个字段进行计算。而普通函数对单个记录进行操作。SQLite 包含了一些标准的聚合函数，例如 SUM()、COUNT()、AVG()。

排序规则是对文本进行比较和排序的方法。也就是说，给定两个字符串，排序规则决定哪个较大或者两个相等。SQLite 中默认的排序规则是 BINARY——它通过 memcmp()逐字节比较字符串。虽然这种排序规则适用于英文（或者其他使用 UTF-8 编码的语言），但是对于使用其他不同编码的语言就不行。因此，SQLite 拥有构建自定义排序规则以处

理这些语言的能力。

本章涉及这三个用户自定义扩展工具以及与之相关的 API 函数。您将看到，用户自定义扩展 API 功能非常强大，非常适于为 SQLite 不同寻常的增强特性。

本章所有的代码都包含在从 Apress 网站（www.apress.com）下载的例子中。本章示例在总目录 examples 下的 ch7 中，我们在介绍示例时会特别注明这些例子对应的源文件。为了清晰起见，图中省略了一些错误处理代码。

此外，为了方便起见，本书很多例子使用了一些有助于简化例子的通用程序库。在介绍这些例子时，本书会指出这些函数并解释它们的功能。通用代码在 examples 文件夹下的 common 中，在构建例子前必须先构建它们。参考 examples 文件夹下的 README 文件，其中有构建通用库的详细说明。

最后，澄清一下，存储类和数据类型是可以通用的。SQLite 内部使用 5 种不同的数据类型或存储类存储数据。如第 4 章所述，这些类型有 integer、float、text、blob 和 null，本章根据上下文情况使用这些类型。我们通常会看到数据类型这个词，因为应用程序中广泛使用这个词，不同背景的人都容易理解。当谈到 SQLite 内部如何处理一个具体值时，存储类看起来比较合适。不管哪种情况，您应该知道它们是相同的，都是用于描述数据的具体形式或者表现形式。

API

实现函数、聚合以及排序规则的基本方法是使用回调函数，并在程序中注册，注册之后，就可以在 SQL 中使用。函数和聚合使用相同的注册函数和类似的回调函数。虽然排序规则使用单独的注册函数，但是该注册函数与函数和聚合的注册函数基本相同，可以认为是用户定义函数的特殊类型。

用户自定义聚合、函数和排序规则的生命周期是短暂的。它们是基于连接注册的，而且不存储在数据库中，需要确保应用程序加载了定制扩展并在连接中注册。有时您可能会把它们当成存储过程看待，而忘记了实际上它们是在数据库之外。它们存在于程序库中，不在数据库中，其生命周期严格限制在程序内。如果没抓住这一点，您可能试图创建引用扩展（扩展不存储在数据库中）的触发器（触发器存储于数据库中），然后试图在 SQLite shell 中调用它们，或者从并不知道该扩展的应用程序中调用它们。这种情

况下，要么触发器报错，要么什么也不做。扩展必须在每一个使用它们的连接上注册，否则，该连接的 SQL 语法根本不知道它们的存在。

注册函数

可以使用 sqlite3_create_function()在连接中注册函数和聚合，如下所示：

```
int sqlite3_create_function(
  sqlite3 *cnx,                    /*连接句柄*/
  const char *zFunctionName,       /*在 SQL 语句中使用的函数聚合名称*/
  int nArg,                        /*参数的格式，-1 表示参数个数不确定*/
  int eTextRep,                    /*编码格式（UTF8，16 等）*/
  void *pUserData,                 /*传递给回调函数的应用程序数据*/
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
int sqlite3_create_function16(
  sqlite3 *cnx,
  const void *zFunctionName,
  int nArg,
  int eTextRep,
  void *pUserData,
  void (*xFunc)(sqlite3_context*, int args, sqlite3_value**),
  void (*xStep)(sqlite3_context*, int args, sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
```


第 6 章的例子中包含了该函数的 UTF-8 和 UTF-16 版本，目的是强调 SQLite 具备支持两种编码的能力。从现在开始，为了简洁起见，我们只使用 UTF-8 版本。

函数 sqlite3_create_function()的参数定义如下：

- **cnx**：数据库连接句柄，函数和聚合的指定连接。若要使用函数，必须在连接上注册。
- **zFunctionName**：将在 SQL 中调用的函数名称，其长度可高达 255 字节。
- **nArg**：自定义函数的参数个数。SQLite 将会强制实施该项，确保提供给自定义函数的格式是正确的。如果值为-1，SQLite 允许参数的个数可变。
- **eTextRep**：首选文本编码。允许值为 SQLITE_UTF8、SQLITE_UTF16、SQLITE_

UTF16BE、SQLITE_UTF16LE 以及 SQLITE_ANY。该选项提示 SQLite 在相同函数的多个实现中选择恰当的编码。SQLITE_ANY 表示“没有偏好”。换句话说，该实现在任何文本编码下同样工作。有疑问时，可以选择 SQLITE_ANY。

- **pUserData**：应用程序数据。该数据可供在 xFunc、xStep 和 xFinal 指定的回调函数中使用。不像其他函数，可以在参数列表中接受 void 指针数据，xFunc、xStep 和 xFinal 中指定的回调函数必须使用特殊的 API 函数获取数据。
- **xFunc**：回调函数。这是 SQL 函数的真实实现。用户自定义函数只需要提供该回调，同时让 xStep 和 xFinal 函数指针为 null。后两个回调函数是专门给用户自定义聚合实现使用的。
- **xStep**：聚合步骤函数。SQLite 每次处理聚合结果集中的一行都要调用 xStep，以便聚合处理该行的相关字段值，并将其包含在聚合计算的结果中。
- **xFinal**：finalize 聚合函数。处理完所有的行后，SQLite 调用该函数进行整个聚合汇总处理，该函数通常是由计算最终值和可选的情况部分组成。

只要编码不同（eTextRep 参数），或参数的个数不同（nArg 参数），同一个函数就可以注册多个版本，SQLite 会自动选择最佳版本。

步骤函数

函数（指自定义函数的回调函数）和聚合的步骤函数是一样的，声明如下：

```
void fn(sqlite3_context* ctx, int nargs, sqlite3_value** values)
```

参数 ctx 是函数/聚合的上下文环境。它保持特殊函数调用的实例状态，通过它可以获得 sqlite3_create_function()提供的应用程序数据参数（pUserData）。您可以使用 sqlite3_user_data()获得用户数据，如下所示：

```
void *sqlite3_user_data(sqlite3_context*);
```

对函数而言，这些数据可以在所有的调用中共享，因此，对函数调用的特定实例而言，它不是唯一的。也就是说，同样的 pUserData 可以在给定函数的所有实例中传递或共享。但是，聚合可以通过 sqlite3_aggregate_context()为每个特定的实例分配状态，如下所示：

```
void *sqlite3_aggregate_context(sqlite3_context*, int nBytes);
```

第一次调用该函数时执行一个特殊的聚合函数，分配 n 字节内存，赋值为零，并与上下文关联。同样，上下文环境中后面的调用（同一个聚合的实例）可以返回这些分配的内存。使用该函数时，聚合可以在调用之间存储状态以便堆积数据（可以考虑一下，这也是聚合的基本目的）。当聚合完成 final()回调时，由 SQLite 自动释放内存。

■注意：您将看到 API 始终以 void 指针形式使用用户数据。这是因为 API 的很多部分都会触发回调函数，在实现所谓的回调函数时，这只是作为一种维护状态的便捷方法。

回调函数的参数 nargs 包含传递给函数的参数个数。

返回值

参数 values 是 sqlite3_value 结构体数组，是 SQLite 实际参数值的句柄。这些值的实际数据可以使用 sqlite3_value_xxx()系列获得，形式如下：

```
xxx sqlite3_value_xxx(sqlite3_value* value);
```


这里的 xxx 是 value 参数返回的 C 语言数据类型。如果您学习过第 6 章，您会发现这些函数与 sqlite3_column_xxx()函数家族有很多相似之处。它们的工作方式相同，只是获取标量和数组值的方式不同。获取标量值的函数形式如下：

```
int sqlite3_value_int(sqlite3_value*);
sqlite3_int64 sqlite3_value_int64(sqlite3_value*);
double sqlite3_value_double(sqlite3_value*);
```

获取数组值的函数如下：

```
int sqlite3_value_bytes(sqlite3_value*);
const void *sqlite3_value_blob(sqlite3_value*);
const unsigned char *sqlite3_value_text(sqlite3_value*);
```

函数 sqlite3_value_bytes()返回 blob 缓冲区中数据的长度。函数 sqlite3_value_blob()返回指向缓冲区数据的指针。有了空间和指针，就可以复制出数据。例如，自定义函数的第一个参数是 blob，想要复制数据时，可以使用如下方式：

```
int len;
void* data;
```

```
len = sqlite3_value_bytes(values[0]);
data = sqlite3_malloc(len);
memcpy(data, sqlite3_value_blob(values[0]), len);
```

正如函数 sqlite3_column_xxx()中的 sqlite_column_type()提供字段类型一样。函数 sqlite3_value_xxx()与函数 sqlite3_value_type()一样，其工作方式相同。该函数声明如下：

```
int sqlite3_value_type(sqlite3_value*);
```

该函数返回如下值之一，这与 SQLite 内核、存储类（数据类型）相符合，具体定义如下：

```
#define SQLITE_INTEGER   1
#define SQLITE_FLOAT     2
#define SQLITE_TEXT      3
#define SQLITE_BLOB      4
#define SQLITE_NULL      5
```

本节覆盖了函数/聚合接口的基本工作内容，现在开始学习一些例子和实际应用。排序规则接口与函数/聚合非常相似，一看便知。

200 函数

我们从简单的例子开始学习自定义函数。首先实现函数 hello_newman()，但是稍有不同。我们的例子中 hello_newman()包含参数，接受一个参数：访问它的人的姓名，如下所示：

```
sqlite > select hello_newman('Jerry') as reply;
reply
------------------
Hello Jerry

sqlite > select hello_newman('Kramer') as reply;
reply
------------------
Hello Kramer

sqlite > select hello_newman('George') as reply;
reply
------------------
```

```
Hello George
```

代码 7-1 显示了基本程序（摘自 hello_newman.c）。为了清晰起见，一些错误处理代码在此删除了。

代码 7-1　hello_newman() 测试程序

```
int main(int argc, char **argv)
{
    int rc; sqlite3 *db;

    sqlite3_open_v2("test.db", &db);
    sqlite3_create_function( db, "hello_newman", 1, SQLITE_UTF8, NULL,
                             hello_newman, NULL, NULL);

    /*在执行 SQL 语句时记录 SQL 日志*/
    log_sql(db,1);

    /*调用函数时，带有一个文本参数*/
    fprintf(stdout, "Calling with one argument.\n");
    print_sql_result(db, "select hello_newman('Jerry')");

    /*调用函数时，带有两个参数，这样会失败，因为该函数注册时写明只接受一个参数*/
    fprintf(stdout, "\nCalling with two arguments.\n");
    print_sql_result(db, "select hello_newman ('Jerry', 'Elaine')");

    /*调用函数时，不带参数，结果也会失败*/
    fprintf(stdout, "\nCalling with no arguments.\n");
    print_sql_result(db, "select hello_newman()");

    /*完成*/
    sqlite3_close(db);

    return 0;
}
```

201

该程序连接数据库，注册函数，然后调用三次。回调函数的实现如下：

```
void hello_newman(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    const char *msg;

    /*产生 Newman 的回复*/
```

```
    msg = sqlite3_mprintf("Hello %s", sqlite3_value_text(values[0]));

    /*设置返回值，让 sqlite 用 sqlite_free()清理 msg */
    sqlite3_result_text(ctx, msg, strlen(msg), sqlite3_free);
}
```

运行程序的结果如下：

```
Calling with one argument.
  TRACE: select hello_newman('Jerry')
hello_newman('Jerry')
--------------------
Hello Jerry

Calling with two arguments.
execute() Error: wrong number of arguments to function hello_newman()

Calling with no arguments.
execute() Error: wrong number of arguments to function hello_newman()
```

第一个调用只有一个参数。因为例子中的函数注册时写明只接收一个参数，所以它成功了。第二个调用使用两个参数，因此失败了。第三个调用没有使用任何参数，同样失败了。

SQLite 的扩展功能是由您决定的，它可以带来很多新函数。本书创建了一些帮助函数，它们有助于您使用 SQLite 的扩展。这些例子包含在从 Apress 网站下载的文件中。许多函数的定义可以在源文件 util.c 中找到。函数 log_sql()只是简单地调用了 sqlite3_trace()，在追踪函数中传递那些带有 TRACE 的 SQL 追踪内容。使用该函数可以看见 SQL 执行中发生了什么。第二个函数是 print_sql_result()，它的声明如下：

```
int print_sql_result(sqlite3 *db, const char* sql, ...)
```

202 这是对 sqlite3_prepare_v2()的简单封装，它将执行 SQL 语句并打印结果。

返回值

例子 hello_newman()引入了新函数 sqlite3_result_text()。该函数是 sqlite_result_xxx()系列的成员，该系列函数是用来为用户自定义函数和聚合返回值的。标量函数如下：

```
void sqlite3_result_double(sqlite3_context*, double);
```

```
void sqlite3_result_int(sqlite3_context*, int);
void sqlite3_result_int64(sqlite3_context*, long long int);
void sqlite3_result_null(sqlite3_context*);
```

这些函数在函数名称中接收一个指定类型的参数（第二个参数指定类型），将其设置为函数的返回值。数组函数如下：

```
void sqlite3_result_text(sqlite3_context*, const char*, int n, void(*)(void*));
void sqlite3_result_blob(sqlite3_context*, const void*, int n, void(*)(void*));
```

这些函数接收数组数据并将该数组设置为函数返回值。它们的常见格式如下：

```
void sqlite3_result_xxx(
    sqlite3_context *ctx,              /*函数上下文*/
    const xxx* value,                  /*数组值*/
    int len,                           /*数组长度*/
    void(*free)(void*));               /*数组清理函数*/
```

这里的 xxx 是特殊的数组类型——void 代表 blob，char 代表 text。再次申明，如果您读过第 6 章，您可能发现这些函数与函数 sqlite3_bind_xxx()相似，函数 sqlite3_bind_xxx()是通用的为函数设置返回值的函数，您也可以将它们当成“绑定返回值”。

数组与内存清理器

与在 SQL 语句中绑定数组值一样，这些函数与函数 sqlite3_bind_xxx()的工作方式相同。它们需要一个指向数组的指针、一个数组的长度（大小），以及指向清理函数的函数指针。清理函数的指针可以像 sqlite3_bind_xxx()中的一样是预定义好的。

```
#define SQLITE_STATIC                ((void(*)(void *))0)
#define SQLITE_TRANSIENT             ((void(*)(void *))-1)
```

SQLITE_STATIC 意味着数组内存驻留在非托管空间，因此，SQLite 不需要数据副本，也不会试图清理它。SQLITE_TRANSIENT 提示 SQLite 数组数据有可能改变，因此，SQLite 需要使用 sqlite3_malloc()为自己复制一份数据。当函数返回时，分配的内存将会释放。第三个选项是传递一个指向如下形式的清理函数的函数指针：

203

```
void cleanup(void*);
```

这种情况下，SQLite 在用户自定义函数完成后，调用清理函数。这也是前面例子中使用的方法。我们用 sqlite3_mprintf()产生前面定义的 hello_newman()回复，该回复在堆

上分配字符串。我们将函数 sqlite3_free()作为清理函数传递给 sqlite3_result_text()，这样当扩展函数完成时，可以释放字符串内存。

错误处理

有时函数会遇见错误，这种情况下，应该设置恰当的返回值。这也是函数 sqlite3_result_error()的作用，其声明如下：

```
void sqlite3_result_error(
    sqlite3_context *ctx,     /*函数上下文*/
    const char *msg,          /*错误消息*/
    int len);                 /*错误消息长度*/
```

它通知 SQLite 出现错误，详细的错误信息包含在 msg 中。SQLite 会终止调用该函数的命令，并将错误信息包含在 msg 中。

返回输入值

有时，可能想将某个参数以相同的形式作为返回值传递回去。不用判断参数的类型，使用相应的函数 sqlite3_column_xxx()获取值，然后用恰当的函数 sqlite3_result_xxx()设置返回值。API 提供函数 sqlite3_result_value()可以一步完成这些操作。该函数的声明如下：

```
void sqlite3_result_value(
    sqlite3_context *ctx,     /*函数上下文*/
    sqlite3_value* value);    /*参数值*/
```

例如，假设您想创建一个可以回显其第一个参数的函数 echo()，实现的命令如下：

```
void echo(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    sqlite3_result_value(ctx, values[0]);
}
```


可以在 SQL 中像如下一样工作：

```
sqlite > select echo('Hello Jerry') as reply;
reply
------------------
Hello Jerry
```

更好的是，echo()对任何存储类都可以做同样的工作：

```
sqlite> select echo(3.14) as reply, typeof(echo(3.14)) as type;
reply       type
----------  ----------
3.14        real
sqlite> select echo(X'0128') as reply, typeof(echo(X'0128')) as type;
reply       type
----------  ----------
(?          blob
sqlite> select echo(NULL) as reply, typeof(echo(NULL)) as type;
reply       type
----------  ----------
            null
```

聚合

聚合比函数涉及的内容要稍多一些。用户在自定义函数中只需要且必须实现一个回调函数就可以，而聚合中必须实现两个，一个是步骤函数计算进行中的值，另一个是 finalize 函数完成最终计算和清理工作。图 7-1 展示了一般的处理过程。与函数一样，恰当的实例可以说明要点。

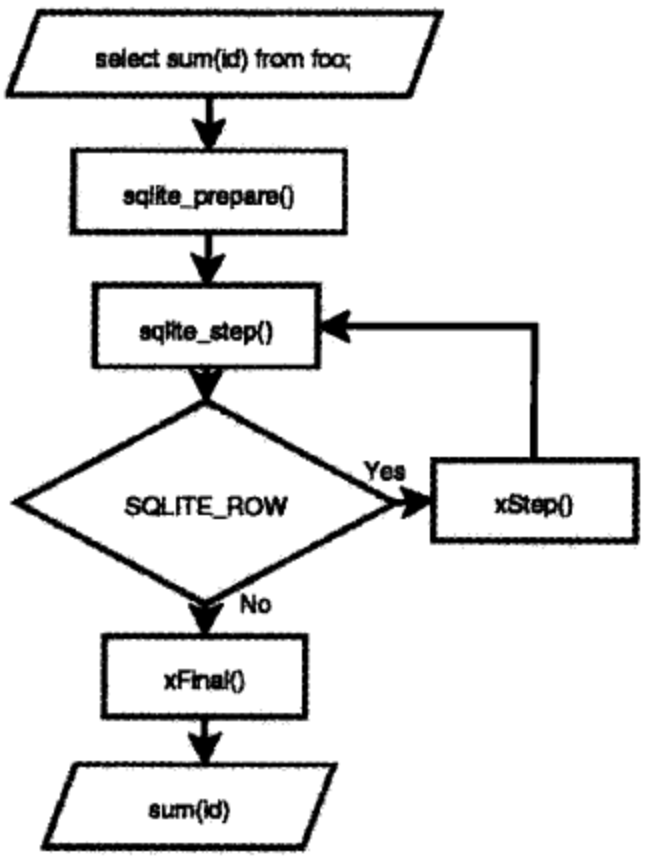


图 7-1　聚合查询处理

注册函数

如前所述，聚合和函数使用同样的注册函数。为了方便起见，再次将函数 sqlite3_create_function()列出：

```
int sqlite3_create_function(
  sqlite3 cnx*,                  /*连接句柄*/
  const char *zFunctionName,  /*在 SQL 语句中使用的函数聚合名称*/
  int nArg,                      /*参数的格式，-1 表示参数个数不确定*/
  int eTextRep,                  /*编码格式（UTF8，16，等）*/
  void *pUserData,               /*传递给回调函数的应用程序数据*/
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
```

206

您可以像注册函数一样做同样的事，但是不用为 xFunc 提供回调函数，可以将 xFunc 设为 null，但是要为 xStep 和 XFinal 提供函数。

实例

说明如何实现聚合的最好方式就是实例。本例中，我们实现聚合 str_agg()。聚合 str_agg()与 C 语言中将两个字符串拼接起来的 strcat()非常相似（技术敏感的人可能注意到，新版的 SQLite 包括类似功能的函数 group_concat()）。在例子中，我们想将聚合分组中涉及的任意数目的字符串进行聚合。考虑到可以用 C 语言实现这样的函数，您有可能在考虑 SQLite 中是否需要这样的字符串拼接聚合。使用聚合 str_agg()的好处是语义清晰，且 SQL 中易于使用，不用为了实现字符串聚合而在 C 语言和 SQL 中来回反复。本例的另一个好处是，向您展示如何实现聚合。代码 7-2 展示了函数 str_agg()（摘自 str_agg.c）。

代码 7-2　str_agg()测试程序

```
int main(int argc, char **argv)
{
    int rc; sqlite3 *db; char *sql;

    sqlite3_open_v2("foods.db", &db);
```

```
    /*打印显示：注册聚合*/
    fprintf(stdout, "Registering aggregate str_agg()\n");

    /*开启 SQL 追踪*/
     log_sql(db, 1);

    /*实际注册聚合*/
     sqlite3_create_function( db, "str_agg", 1, SQLITE_UTF8, db,
                              NULL, str_agg_step, str_agg_finalize);

    /*测试聚合*/
    fprintf(stdout, "\nRunning query: \n");
    sql = "select season, str_agg(name, ', ') from episodes group by season";
    print_sql_result(db, sql);

    sqlite3_close(db);

    return 0;
}
```

注意，我们的步骤函数叫做 str_agg_step()，finalize 函数叫做 str_agg_finalize()。该聚合函数注册成只接收一个参数。

步骤函数

207

代码 7-3 展示了 str_agg_step()函数。

代码 7-3　步骤函数

```
void str_agg_step(sqlite3_context* ctx, int ncols, sqlite3_value** values)
{
  SAggCtx *p = (SAggCtx *) sqlite3_aggregate_context(ctx, sizeof(*p));

  static const char delim [] = ", ";

  char *txt = sqlite3_value_text(values[0]);

  int len = strlen(txt);

  if (!p->result) {
    p->result = sqlite_malloc(len + 1);
    memcpy(p->result, txt, len + 1);
    p->chrCnt = len;
```

```
  }
  else
  {
    const int delimLen = sizeof(delim);
    p->result = sqlite_realloc(p->result, p->chrCnt + len + delimLen + 1);
    memcpy(p->result + p->chrCnt, delim, delimLen);
    p->chrCnt += delimLen;
    memcpy(p->result + p->chrCnt, txt, len + 1);
    p->chrCnt += len;
  }
}
```

SAggCtx 是本例中的结构体，定义如下：

```
typedef struct SAggCtx SAggCtx;
struct SAggCtx {
  int chrCnt;
  char *result;
};
```

该结构体充当聚合迭代（对步骤函数的调用）之间的状态，维持不断增长的拼接文本和字符数。

聚合上下文

第一项工作就是为给定的聚合实例获取结构体，这是通过函数 sqlite3_aggregate_context()完成的。如前所述，对该函数的第一次调用将会为该上下文分配数据，之后的调用直接获取。聚合完成后（在调用 str_agg_finalize()后），结构体内存自动释放。注意，如果系统内存较少，sqlite3_aggregate_context()可能返回 null。为了代码清晰，本例中省略了这种错误检查。实际上，真正的实现必须总是检查 sqlite3_aggregate_context()的结果，如果结果为 null，需要触发 sqlite3_result_error_nomem()。在函数 str_agg_step()中，可以使用函数 sqlite3_value_text()从第一个参数获取被聚合的文本值。然后将值添加到结构体 SAggCtx 的 result 成员中，该结构体存储中间拼接字符串。再次注意，如果 SQL 函数的参数是 null，或者系统内存较少，sqlite3_value_text()可能返回 null。为了避免段错误（segfault），实际实现时应该检查这种情况。

208

Finalize 函数

结果集中每一个记录都触发一次 str_agg_step()调用。在处理完最后的记录后，SQLite

调用 str_agg_finalize()，str_agg_finalize()的实现如代码 7-4 所示。

代码 7-4　finalize 函数 sum_agg()

```
void str_agg_finalize(sqlite3_context* ctx)
{
  SAggCtx *p = (SAggCtx *) sqlite3_aggregate_context(ctx, sizeof(*p));
  if( p && p->result ) sqlite3_result_text(ctx, p->result, p->chrCnt, sqlite_free);
}
```

函数 str_agg_finalize()像用户自定义的回调函数一样工作。它获取结构体 sAggctx，如果该结构体不为 null，通过函数 sqlite3_result_text()将聚合的返回值设置为结构体中存储的成员 result。如果系统内存较小，或从未调用步骤函数，p->zResult 可能是 null，sqlite3_result_text()可能返回 null。前一种情况中，SQLite 会返回错误，实际上与是否调用 sqlite3_result_text()无关。后一种情况中，因为聚合从未触发函数 sqlite3_result_xxx()，聚合结果将会是 null。

结果

程序产生如下输出（省略部分内容）：

```
Registering aggregate str_agg()

Running query:
TRACE: select season, str_agg(name, ', ') from episodes group by season
…

season   str_agg(name, ', ')
------   ------------------------------------------------------------------
0        Good News Bad News
1        Male Unbonding, The Stake Out, The Robbery, The Stock Tip
2        The Ex-Girlfriend, The Pony Remark, The Busboy, The Baby Shower, …
3        The Note, The Truth, The Dog, The Library, The Pen, The Parking Garage …
…
```

如您所见，select 语句从 episode 表中获取数据，聚合 str_agg()拼接各个季度的系列名称。您可以将其想象成电子表格中季度统计或其他工具。

209

简单地说，这就是聚合的概念。它们与函数非常类似，以至于了解函数后就不需要再次介绍聚合了。

排序规则

如前所述，排序的目的就是对事物进行分类比较。开始之前，回想一下 SQLite 显示类型模式允许在一个字段中存在各种数据类型。例如，考虑如下例子（文件 collate1.sql）：

```
.headers on
.m column
create table foo(x);
insert into foo values (1);
insert into foo values (2.71828);
insert into foo values ('three');
insert into foo values (X'0004');
insert into foo values (null);

select quote(x), typeof(x) from foo;
```

将这些命令在 SQLite 命令行下执行，结果如下：

```
C:\temp\examples\ch7> sqlite3 < collate.sql
quote(x)    typeof(x)
----------  ----------
1           integer
2.71828     real
'three'     text
X'0004'     blob
NULL        null
```

在字段 x 中使用 SQLite 的原生存储类，这时问题产生了：排序时如何排列这些不同类型的值？也就是说，开始讨论如何对字符串进行排序之前，我们需要先看看不同类型的数据排序有何不同。

SQLite 对结果集中的字段进行排序时（SQLite 使用比较操作符如< 或 >=在字段内对值进行比较），第一件事就是根据存储类对字段值进行排列。然后在每种存储类中，根据该类指定的方法进行排序。存储类进行排序的顺序从前往后如下所示：

1. NULL 值
2. INTEGER 和 REAL 值
3. TEXT 值

4．BLOB 值

现在我们修改前面的 select 语句，加上 order by 子句。修改后的 SQL 语句如下：

210

```
select quote(x), typeof(x) from foo order by x;
```

运行查询（位于 collatez.sql），验证排序结果：

```
C:\temp\examples\ch7> sqlite3 < collate.sql
quote(x)     typeof(x)
----------   ----------
NULL         null
1            integer
2.71828      real
'three'      text
X'0004'      blob
```

NULL 是第一位，紧接着是 integer 和 real（按照数字顺序），其次是 text，最后是 blob。

SQLite 对每种存储类使用具体的排序算法。NULL 很特别，它没有顺序。数字是按照数字值大小进行排序的；整型和浮点型基于它们各自的数量值进行比较；BLOB 按照二进制值进行排序。最后，文本就是我们要讨论的排序规则的重点。

排序法定义

排序规则就是比较字符串的方法，排序规则的专业定义[①]如下：

与逐个比特比较数字代码相反，排序规则使用语言敏感的规则对文本进行比较。

一般来讲，排序规则必须处理字符串和字符的排序、比较以及排列。排序规则通常使用排序序列。排序序列就是已经排好序的字符列表，通常都有数字编码，以便记忆和使用。该列表也是用来判断字符是如何排序、比较以及分类的。给定任何两个字符，排序规则（列表）必须判断出哪个在前，或者两个相等。

排序规则如何工作

通常来讲，排序方法对将要比较的两个字符串进行切分，通过排序序列逐一比较每

① 参考：http://www.ibm.com/developerworks/library/glossaries/unicode.html.

一个字符。它们从左到右比较字符（见图 7-2），例如，字符串 'jerry' 和 'jello' 首先比较
211 各自的第一个字符（本例中两个都是 j），如果某个字符的数字值根据排序序列大于另外一个，比较结束：有较大字符的字符串是较大者，不需要进一步比较。另一方面，如果两个字符的值相等，继续处理各自字符串后面的字符，第二个、第三个，等等，直到发现不同（如果无法发现不同，这种情况可以认为两个字符串是相等的）。如果字符串比较完所有的字符还未分出大小，那么由排序规则决定这种情况如何处理。

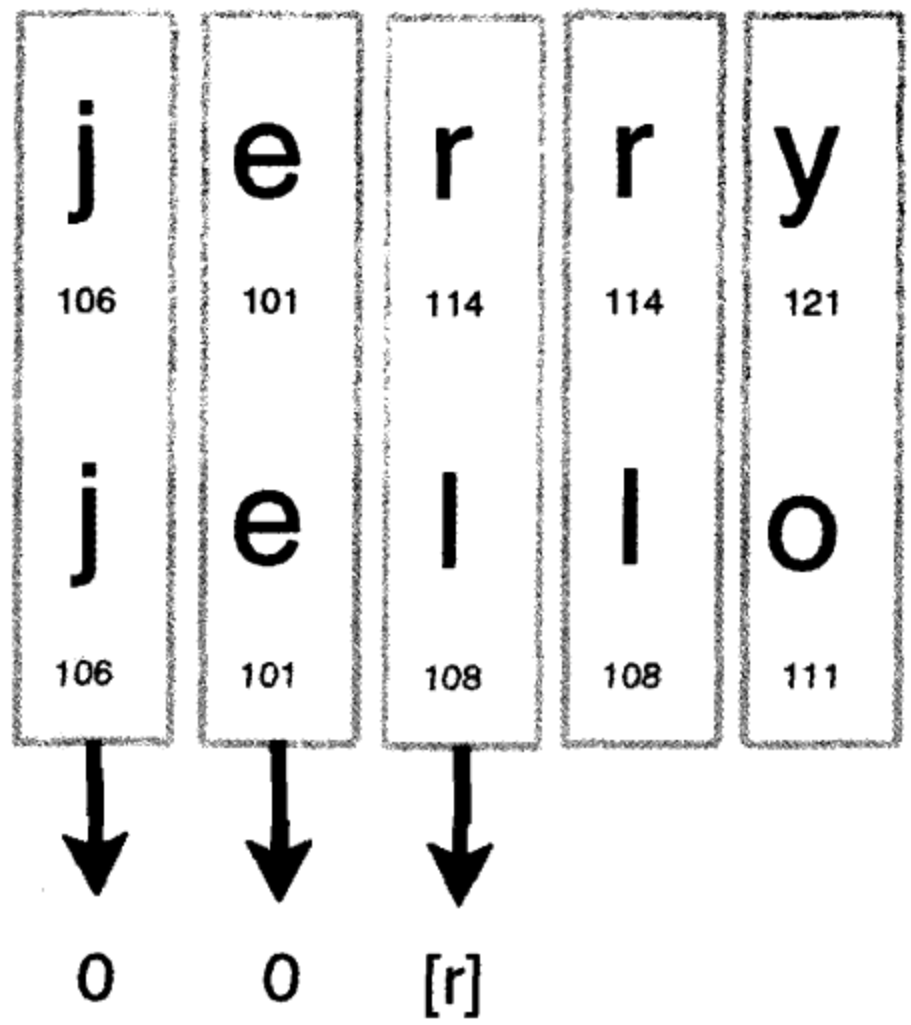


图 7-2　binary 排序规则

本例中，排序规则是 ASCII 字符集，图中显示了每个字符的 ASCII 值。使用刚描述的排序规则，字符串比较进行到第三个字符时，'jerry'胜出，'jerry'中的 r 值是 114，而'jello'中的 l 值是 108。根据 ASCII 排序规则，'jerry' > 'jello'。

如果'jerry' 变成'Jerry',，那么'jello'胜出，因为大写的 J 是 74，而小写的 j 是 104，比较会在第一个字符处结束，并判定'jello'胜出，即'jello' > 'Jerry'。

继续这个例子，如果使用不同的排序规则，比如大小写不敏感的排序规则。那么'Jerry'和'jello' 比较的结果是 'Jerry'胜出，因为大写的 J 和小写的 j 在排序规则中是相等的，'jello' 中小写的 j 不占优势。 212

现在对排序规则已经有了总体的了解。序列仅仅是序列，最终是要由排序方法决定排序顺序而不是排序序列的。排序方法不是必须使用排序序列的，它们可以做任何想要的比较，不管看起来有多奇怪或者疯狂。

综上所述，排序规则就是字符串的比较方法。排序序列就是字符的比较方式。最终，在 SQLite 中字符串是如何比较的，是由这两者共同决定的。因此，排序方法和排序序列都可以看做是字符串比较。总之，它们都是关于字符串比较的。创建定制的排序规则时，SQLite 处理两个字符串 a 和 b，排序规则返回是否 a 大于、等于或小于 b。就是这样简单而实际。

标准排序规则类型

SQLite 中唯一内置的排序规则 binary 使用标准的 C 函数库 memcmp()比较两个字符串。binary 排序规则对于英文文本适用很好。但对于其他语言，可能需要其他的排序规则。SQLite 还提供了另一种排序规则 nocase，它是大小写不敏感的。

排序规则是通过表字段或索引定义以及在查询中指定的方法和字段关联起来的。例如，在 foo 中创建大小写不敏感的 bar 字段，可以使用如下定义：

```
create table foo (bar text collate nocase, baz integer);
```

从那时起，SQLite 处理 bar 字段时，就会使用 nocase 排序规则。如不想将排序规则附加到数据库对象上，而是在需要时在查询中指定，可以直接在查询中指定它们，例如：

```
select * from foo order by bar collate nocase;
```

SQLite 也可以对 cast()表达式的结果应用排序规则，cast()表达式可以将非文本数据（这些数据通常没有文本的排序规则）转换为文本类型数据。我们可以将 bar 数字值转换为文本，并像下面这样应用 nocase 排序：

```
select cast(baz) collate nocase as baz_as_text from foo order by baz_as_text;
```

虽然上面的例子有些不自然，但是这有助于查询处理，特别是将那种混合存储类型的数据转化为文本的情况。

简单例子

要了解排序规则，我们从简单例子开始，从整体上有所掌握。先实现一个名为 length_first 的排序规则，length_first 接收两个字符串并依据字符串长度决定哪个更大。想象一下该排序规则可以应用的场合，例如，Jerry 在熟食店，要求吃那些名字较短的食物——Kramer 坚持要吃那些名字很长的食物。

213 函数 sqlite3_create_collation_v2()在 SQLite 中注册排序规则，该函数的声明如下：

```
int sqlite3_create_collation_v2(
  sqlite3* db,                    /*数据库句柄*/
  const char *zName,              /* SQL 语句中使用的排序规则名称*/
  int pref16,                     /*编码格式*/
  void* pUserData,                /*应用程序数据*/
  int(*xCompare)(void*,int,const void*,int,const void*)
  void(*xDestroy)(void*)
);
```

该函数与函数 sqlite3_create_function()非常相似。第一个参数是标准的数据库句柄，第二个是在 SQL 中使用的排序规则名称（与函数/聚合名称类似），然后是编码方式（编码方式从属于传递给比较函数时被比较字符串的格式）。

> **注意**：与聚合和函数一样，单独的排序比较函数可以用同一个排序规则名称为 UTF-8、UTF-16LE 和 UTF-16BE 编码注册多次。SQLite 将根据要比较的字符串自动选择最佳的比较函数。

应用程序数据指针是另一个 API 的主要支撑，该指针传递给回调函数。函数 sqlite3_create_collation_v2()基本上是删减版的 sqlite3_create_function()函数，函数 sqlite3_create_function()输入两个参数（两个参数都是文本），返回一个整型值。函数 sqlite3_create_collation_v2()去掉了聚合回调函数指针。

Compare 函数

参数 xCompare 在此是一个新词，它指向实际解析文本值的比较函数。比较函数格式必须如下：

```
int compare( void* data,          /*应用程序数据*/
             int len1,            /*字符串 1 的长度*/
```

```
        const void* str1,     /*字符串 1*/
        int len2,             /*字符串 2 的长度*/
        const void* str2)     /*字符串 2*/
```

如果第一个字符串较小于、等于或大于第二个字符串，该函数应该返回负值、零或者正值。

如上所述，length_first 排序有些空泛。代码 7-5 给出了它的实现。

代码 7-5　length_first 排序函数

214

```
int length_first_collation( void* data, int l1, const void* s1,
                                int l2, const void* s2 )
{
    int result, opinion;

    /*比较长度*/
    if ( l1 == l2 ) result = 0;

    if ( l1 < l2 ) result = 1;

    if ( l1 > l2 ) result = 2;

    /*形成一种观点:s1 真的小于或等于 s2 吗? */
    switch(result) {
        case 0: /*长度相等，按照字母数字顺序进行排序*/
            opinion = strcmp(s1,s2);
            break;
        case 1: /*字符串 s1 较短*/
            opinion = -result;
            break;
       case 2: /*字符串 s2 较短*/
            opinion = result
            break;
       default: /*假设长度相等，以防万一（如果 result 等于其他值）*/
            opinion = strcmp(s1,s2);
    }

    return opinion;
}
```

测试程序

接下来就是如何在程序中使用该函数。代码 7-6 给出了示范（摘自 length_first.c）。

代码 7-6　leng_first 排序测试程序

```
int main(int argc, char **argv)
{
    char *sql; sqlite3 *db; int rc;

    sqlite3_open("foods.db", &db);

    /*注册排序规则*/
    fprintf(stdout, "1. Register length_first Collation\n\n");
    sqlite3_create_collation_v2( db, "LENGTH_FIRST", SQLITE_UTF8, db,
                                 length_first_collation, length_first_collation_del );

    /*开启 SQL 日志记录*/
    log_sql(db, 1);

    /*测试默认的排序规则*/
    fprintf(stdout, "2. Select records using default collation.\n");
    sql = "select name from foods order by name";
    print_sql_result(db, sql);

    /*测试排序规则 length_first */
    fprintf(stdout, "\nSelect records using length_first collation. \n");
    sql = "select name from foods order by name collate LENGTH_FIRST";
    print_sql_result(db, sql);

    /*结束*/
    sqlite3_close(db);

    return 0;
}
```

215

结果

运行程序产生如下结果（为节省空间，此处省略部分内容）。

```
1. Register length-frist collation

2. Select records using default collation.
  TRACE: select name from foods order by name
```

```
name
-----------------
A1 Sauce
All Day Sucker
Almond Joy
Apple
Apple Cider
Apple Pie
Arabian Mocha Java (beans)
Arby's Roast Beef
Artichokes
Atomic Sub
...
Select records using length_first collation.                    216
  TRACE: select name from foods order by name collate LENGTH_FIRST
issue
-----------------
BLT
Gum
Kix
Pez
Pie
Tea
Bran
Duck
Dill
Life
...
```

测试程序从注册排序规则 length_first 开始。然后使用 SQLite 默认的 binary 排序规则处理 select 记录，select 返回按照字符顺序排序的记录。接着使用 length_first 排序规则，再次执行 select 语句。正如结果所示，首先按照长度排序，然后按字符顺序排序。

按需排序

SQLite 提供一种直到实际需要时才注册排序的延迟方式。因此，如果不确定您的程序是否需要一些像 length_first 的排序规则，您可以使用函数 sqlite3_collation_needed()，将注册延迟到最后可能需要的时刻（就好似神秘的餐后甜点一样）。只需要提供回调函

数，SQLite 会在需要时注册未知排序规则，并给定名称。打个比方说，这就像回调函数的回调一样，对 SQLite 说："您好，如果您被问到使用某个您不认识的排序规则时，调用该函数，并注册未知排序序列，这样您就可以继续工作。"

函数 qlite3_collation_needed()的声明如下：

```
int sqlite3_collation_needed(
  sqlite3* db,                    /*连接句柄*/
  void* data,                     /*应用程序数据*/
  void(*crf)(void*,sqlite3*,int eTextRep,const char*)
);
```

参数 crf 指向将注册未知排序规则的函数，为清晰起见，其定义如下：

```
void crf( void* data,           /*应用程序数据*/
          sqlite3* db,          /*数据库句柄*/
          int eTextRep,         /*编码格式*/
          const char*)          /*排序规则名称*/
```

函数 crf()的参数 db 和 data 是传递给 sqlite3_collation_needed()的第一和第二个参数值。

217 总结

现在，您已经了解到用户自定义函数、聚合和排序规则相当有用。使用扩展 C API 向 SQLite 核心库添加这些自定义函数是对 SQLite 核心库的重要补充。因此，您可以深挖和修改 SQLite 的核心内容，甚至可以说 SQLite 提供了友好的、易于使用的、可以更大范围扩展和定制自身的用户接口，特别是和 SQLite 中已有功能结合时。

下一章中，您会看到很多语言如何使用这一功能。它们使用 C API 为各种语言实现很多扩展和各种各样的功能。

第 8 章

语言扩展

SQLite 是用 C 语言编写的，并且提供 C API，这在某种程度上就意味着 C 语言是 SQLite 的“母语”。但是，开源社区提供了许多 SQLite 扩展包，使得 SQLite 可以应用于其他编程语言和类库，比如 Perl、Pthon、Ruby、Java、.NET/C#、Qt 和 ODBC。在很多情况下，针对一种语言有很多扩展包可供选择，这些扩展包都是不同的程序员为了满足不同需求而开发的。

许多扩展包都符合多种 API 标准。例如，某个 SQLite Perl 扩展包遵循 Perl DBI 标准——该标准是 Perl 访问数据库的接口标准。类似的，Python 扩展包遵循 Python DB API 规范，Java 扩展包遵循 JDBC 标准。但是这些不同语言的扩展包无论使用哪种特殊的 API，本质上它们都是在同样的 SQLite C API 基础上进行扩展的，在第 5～7 章中已经使用过 SQLite C API。从某种程度上说，所有的扩展包都反映了 SQLite 的 C 语言设计。有些扩展包既提供了符合它们自身 API 标准的接口，又提供了能够更好地反应 SQLite API 设计的接口。因此，在这种情况下，用户可以针对需要，选择最合适的接口。

总体而言，所有的扩展包不管它们的 API 有多么特殊，它们的底层机制架构都是相似的，都遵循相似的模式。一旦理解了这种模式，所有的接口在很多方面看起来都是相似的。在这种模式中，包含某种类型的连接到数据库的连接对象，另外，还包含某种代表 SQL 查询的游标对象，借助该游标对象既可以执行数据库查询命令，又可以遍历查询结果。这种模式从概念上很容易理解，其细节可以参考前面章节中介绍过的 sqlite3 结构体和 sqlite3_stmt 结构体，这两种结构体在之前章节中已经详细解释并举例过。C API 的规则也可应用于扩展语言的扩展包。

本章将讲解 6 种流行语言的扩展包机制：Perl、Python、Ruby、Java、Tcl 和 PHP。目的是提供一个简单明了的介绍，使用户可以在不同的编程语言中快速使用 SQLite。需

要指出的是，我们不会具体介绍这些语言本身，所以需要用户对这些语言（Perl、Python、Ruby 等）有所了解。

在介绍这几种语言时，我们将覆盖如下常用的知识框架，该框架是人们使用扩展语言包时常见问题的总结：

- 与数据库连接。
- 执行查询语句。
- 使用绑定参数。
- 实现用户自定义的函数或聚合。

上面这些内容组成了用户在使用 SQLite 扩展包时常用的绝大多数功能。用户可能会面临两难的抉择：是应该在 SQLite 中开发自定义方法，还是使用所选择的编程语言中已有的方法实现某种功能？尽管我们不能替用户做出决定，但是我们会提供有助于用户做出选择的相关知识和专业意见。

不是所有语言的扩展包都支持上面列出的内容。比如，有些扩展包不提供绑定参数。另外，有些扩展包还提供这里没有列出的一些功能。我们的目的是提供一种统一、直接的方法，使用户可以在各种编程语言中使用 SQLite。一旦对扩展包有了了解，那么其他扩展包中的特殊功能也很容易掌握。

本章会介绍不同语言的扩展包是如何使用 C API 各个部分的，以帮助用户理解底层是如何工作的。如果没有阅读过前面章节中关于 C API 的介绍，强烈推荐先阅读它们。无论使用何种编程语言，理解 SQLite 是如何工作的都将有助于更好地利用扩展包和编写更好的代码。了解 C API 之后，用户会知道如何选择最适合的查询方法，理解事务的范围，以及懂得怎样预测、处理和避免死锁等问题。

选择一种扩展语言

目前，流行的编程语言有很多，当要选择其中一种语言与 SQLite 绑定时，总是眼花缭乱，不知选择哪种好，因为选择适合的扩展语言不只是技术问题。为了帮助大家选择合适的扩展语言，我们提供了一些选择原则：

- 支持最新的 SQLite 3 版本。
- 具备完善的文档。

- 稳定性。
- 可移植性。

有很多扩展语言满足这些条件。这种情况下，选择哪种语言基本取决于个人偏好。我们的目的不是要告诉用户选择哪个具体的扩展包，而是想传授相关的概念，使您能够准确选择任一种语言扩展包，并把对应的语言扩展包添加到 SQLite 中使用。

我们选择某个扩展包的理由可能与您不同。以下几个因素也是在选择时需要考虑的。

- 许可证和许可证的兼容性：扩展包的许可证将直接影响用户怎样使用它。扩展包是开源的吗？如果是，它的许可证是什么？如果在为商业软件编写代码，那么必须考虑这个因素。
- 数据类型映射：扩展包是如何把 SQLite 的存储类型映射为编程语言本身类型的。是不是所有的值都作为文本类型返回？是否疯狂地使用类型转换机制？如果不是，是否有简单的方法判断这种映射？能否控制这种映射的结果？
- 查询方法：C API 支持三种不同的查询方法。扩展包使用的是哪种？理想情况是对三种方法都支持。
- API 覆盖度：扩展包是否能覆盖其他不能很容易映射到标准数据库接口的方法？例如，是否允许调用 SQL 追踪函数，操作一些与安全相关的函数？是否实际需要这些功能？
- 链接和发布：扩展包是否能够很容易地与任何一个版本的 SQLite 捆绑使用？扩展包里是否已经包含了某个版本的 SQLite？它是如何与 SQLite 关联的，是使用共享库，还是静态链接到某个版本的 SQLite？如果需要升级 SQLite，又会怎样，升级是否简单？会不会遇到程序崩溃的情况？ 221

还有其他一些因素也需要考虑，比如社区论坛的支持、邮件组的活跃度、维护者的支持以及回复、回归测试、代码质量等很多因素。所有这些因素都可能对用户决定选择哪个扩展包有重要的作用。只有用户自己能回答这些问题。

SQLite wiki 网站提供了各种编程语言接口的详细代码，可以在 www.sqlite.org/cvstrac/wiki?p=SqliteWrappers 中找到它们。本章要介绍的所有接口都可以在这个列表中找到，所有的实例源代码都可以在 examples zip 文件的 ch8 目录中找到，examples zip 可以在 Apress 网站上下载。

Perl

Perl 有两个 SQLite 扩展包。一个是建立在 Perl DBI 标准基础上的，负责与数据库连接，因此，如果您了解 DBI，就会很快掌握这个扩展包的使用。如果您不熟悉 Perl DBI，可以参考 CPAN 网站上的文档 http://search.cpan.org/search?module=DBI。另一个扩展包是基于 SWIG 的。下面我们介绍 DBI 接口。

安装

目前，SQLite DBI 驱动的版本是 DBD-SQLite-1.29，可以借助 CPAN 安装它，或者手动从源代码编译建立。使用 SQLite DBD 模块的先决条件是具备 Perl、C 编译器以及 DBI 模块。最新版本的 SQLite DBI 驱动还需要使用 YAML（“Yet another markup language”或者“YAML Ain’t Markup Language”）来设置各种配置文件。借助 CPAN 安装 DBD::SQLite 时，需要在命令行运行如下命令启动 CPAN：

```
fuzzy@linux $ cpan

cpan> install DBD::SQLite

... lots of CPAN install messages ...
```

如果在 Linux/UNIX 中从源文件安装，需要切换到一个临时目录，并进行如下操作：

```
fuzzy@linux $ wget http://search.cpan.org/CPAN/authors/id/A/AD/ADAMK/DBD-SQLite-1.29.tar.gz

... lots of wget messages ...

fuzzy@linux $ tar xzvf DBD-SQLite-1.29.tar.gz

... lots of tar messages ...

fuzzy@linux $ cd DBD-SQLite-1.29

fuzzy@linux DBD-SQLite-1.29 $ perl Makefile.PL

... lots of perl messages ...
```

```
Checking installed SQLite version...
Looks good
fuzzy@linux DBD-SQLite-1.29 $ make install

... more perl messages ...
```

SQLite Perl 扩展包本身就包含 SQLite 3 二进制文件，因此，没有必要事先编译和安装 SQLite。SQLite 已经嵌入在扩展包里了。

连接

在 Perl 中使用 DBI 函数 available_drivers()检查 SQLite 驱动是否已经安装：

```
use DBI;
print "Drivers: " . join(", ", DBI->available_drivers()), "\n";
```

可以使用标准的 DBI::connect 函数连接到数据库，如下所示：

```
use DBI;
my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });
$dbh->disconnect;
```

第二个和第三个参数分别对应驱动器和数据库的名字，第三个和第四个参数分别对应用户名和密码，用户名和密码对 SQLite 并不适用。

该函数返回一个表示与数据库连接的数据库句柄对象。在内部，这个返回值对应一个简单的 `sqlite3` 结构体。可以在数据库名字前面加上`:memory:`，表示创建内存数据库。这种情况下，所有的数据库表和其他数据库对象在整个程序运行期间都将驻留在内存中，当与数据库的连接关闭时销毁数据库对象。可以使用数据库句柄的 `disconnect` 函数关闭数据库。

查询处理

数据库查询也是通过标准的DBI接口执行的。可以使用数据库连接句柄的prepare()、execute()、fetch()或者 selectrow()函数查询。在 DBI 手册里可以找到所有这些函数的文档。代码 8-1 显示了一个查询的例子。

代码 8-1　在 Perl 中执行查询

```
use DBI;
# 连接数据库
my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });

# 准备语句
my $sth = $dbh->prepare("select name from foods limit 3");

# 执行
$sth->execute;

# 以可读方式打印标题
print "\nArray:\n\n";

# 递归编历结果并打印
while($row = $sth->fetchrow_arrayref) {
      print @$row[0] . "\n";
}

# 做同样的事情，不过这次使用 hashref
print "\nHash:\n\n";
$sth->execute;
while($row = $sth->fetchrow_hashref) {
  print @$row{'name'} . "\n";
}

# 结束语句
$sth->finish;

# 断开连接
$dbh->disconnect;
```

223

从内部看，prepare()函数对应 sqlite3_prepare_v2()函数。类似地，execute()函数调用第一个 sqlite3_step()函数，它会自动判断出是否有数据返回。如果没有数据（例如，非 select 语句），它会自动终结查询。函数 fetchrow_array()、fetchrow_hashref()以及 fetch()都在调用 sqlite3_step()，每次返回结果集中的一行，直到返回所有的结果。

另外，使用数据库对象的 do()函数可以用一步运行非 select 语句。代码 8-2 展示了内存数据库中如何使用该方法。

代码 8-2　do()函数

```
use DBI;
my $dbh = DBI->connect("dbi:SQLite:dbname=:memory:", "", "", { RaiseError => 1 });
$dbh->do("create table cast (name)");
$dbh->do("insert into cast values ('Elaine')");
$dbh->do("insert into cast values ('Jerry')");
$dbh->do("insert into cast values ('Kramer')");
$dbh->do("insert into cast values ('George')");
$dbh->do("insert into cast values ('Newman')");

my $sth = $dbh->prepare("select * from cast");
$sth->execute;

while($row = $sth->fetch) {
    print join(", ", @$row), "\n";
}

$sth->finish;
$dbh->disconnect;
```

224

如果还没有调用过 sqlite3_finalize()函数，语句句柄对象的 finish()方法会在查询对象上调用 sqlite3_finalize()函数。上面程序的运行结果如下：

```
Elaine
Jerry
Kramer
George
Newman
```

参数绑定

参数绑定遵循 DBI 规范中定义的方法。尽管 SQLite 既支持位置参数，也支持命名参数，但是 Perl 接口只支持位置参数，这点可以从代码 8-3 中看出。

代码 8-3　Perl 中的参数绑定

```
use DBI;

my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });
```

```
my $sth = $dbh->prepare("select * from foods where name like :1");
$sth->execute('C%');

while($row = $sth->fetchrow_hashref) {
    print @$row{'name'} . "\n";
}

$sth->finish;
$dbh->disconnect;
```

函数 execute()将 prepare()函数中定义的参数值作为参数。在前面的例子中用到的 foods 数据库上运行这段代码，将返回 73 行结果（取决于在前面章节中 chocolate bobka 被返回的次数）。

用户自定义函数

用户自定义函数和聚合通过使用驱动的私有函数来实现。它们遵循的规则与 SQLite C API 中对应的函数非常相似。自定义函数使用如下的私有函数进行注册：

225 `$dbh->func( $name, $argc, $func_ref, "create_function" )`

这里的参数$name 指 SQL 函数的名字，$argc 指参数的个数，$func_ref 指向提供实现的 Perl 函数。代码 8-4 用 Perl 中的一个 hello_newman()的例子解释该函数。

代码 8-4　Perl 中的 hello_newman()

```
use DBI;

sub hello_newman {
    return "Hello Jerry";
}

# 连接
my $dbh = DBI->connect( "dbi:SQLite:dbname=foods.db", "", "",
                        { RaiseError => 1 });

# 注册函数
$dbh->func('hello_newman', 0, \&hello_newman, 'create_function');

# 调用刚注册的函数
print $dbh->selectrow_arrayref("select hello_newman()")->[0] . "\n";
```

```
$dbh->disconnect;
```

如果提供的参数个数是-1，那么 hello_newman()函数的参数个数是不确定的。

聚合

用户自定义聚合就如同函数包一样的执行，每个包都执行一个步骤函数和一个最终函数。该步骤函数在每一行被选择时调用，提供给该函数的第一个值是运行环境，这是个常量，在调用对象与函数间维护。其他值都是 SQL 语句提供给聚合函数的参数。最终函数需要一个与步骤函数一样的运行环境。下面的例子实现了一个简单的 SUM 聚合语句，名为 perlsum。这个聚合语句在 perlsum.pm 包中执行，如代码 8-5 所示。

代码 8-5　The perlsum()聚合

```
package perlsum;

sub new { bless [], shift; }

sub step {
<A HREF="http://rubyforge.org/projects/sqlite-ruby" CLASS="URL"> my ( $self, $value ) =
@_;
    @$self[0] += $value;
}

sub finalize {
    my $self = $_[0];
    return @$self[0];
}

sub init {
    $dbh = shift;
    $dbh->func( "perlsum", 1, "perlsum", "create_aggregate" );
}

1;
```

init 函数用来在数据库连接中注册聚合函数。

Python

在 Python 中有许多使用 SQLite 的方法，这些方法可以分为两种类型：第一类是针对 SQLite 扩展包的方法，比如 PySQLite 和 APSW，它们直接利用包的接口调用底层的 SQLite API（尽管不同函数调用的深度不同）。第二类是包含针对所有框架的方法，比如 SQL Alchemy，这类函数是完全按照对象-关系规则设计的，它不仅提供对底层数据库的 API 访问接口，还在设计中遵循对象-关系的原则，使用关系数据库代码。

根据最近一次统计，Python 至少包含 9 种不同的为 SQLite 服务的包和对象-关系框架。仅介绍这几个包和对象就可以写一本书了。这里，我们只介绍一种比较流行的包 PySQLite，因为它的可扩展性最好，并且与 Python DB API 相映射。

本书第一版的作者 Michael Owens 在 2002 年开始编写代码，完成了 PySQLite 的第一版，当时使用的是 SQLite2.x API。后来由 Gerhard Haring 接手这个项目，几乎重写了所有的代码，产生了 PySQLite 的第二版，第二版版本支持 SQLite 3。随后又进行了多次修改和改进，现在它的名称是 pysqlite（所有都是小写字母）。可以在 Google Code（http://code.google.com/p/pysqlite/）中找到 PySQLite 项目的相关信息，源代码和 Windows 上的可执行文件可供下载。

安装

Python 2.5 以后的版本都把 pysqlite 作为 Python 的一个标准模块，在这个模块中使用 SQLite 库时，调用的是 sqlite3 模块。有时还需要获取和编译/使用独立的 pysqlite 版本，这样更接近 Python 的使用，可以调试代码错误和使用新功能。

为了能够使用 pysqlite 包中的源代码，我们推荐 Windows 用户直接使用已经提供的二进制文件，不要自行编译。UNIX、Linux、Mac OS X 以及其他系统的用户可以利用 Python 的 distutils 包编译和安装 pysqlite，或者编译本地的 rpm（deb）安装包查看源码。

227 编译 PySQLite 要求有 C 编译器、Python 2.3 或之后的版本以及 SQLite3.1 或之后的版本。解压源代码后，利用下面的命令编译代码：

```
Python setup.py build
```

distutils 应该能够识别出系统中的任何文件，否则，会返回错误信息。此时，需要

修改主目录下的 setup.cfg 文件，指出所需要文件的路径。如果一切正常，会看到输出“正在创建类库...”，这表明已经成功编译了代码。下一步安装类库：

```
python setup.py install
```

这一步是不应该有错误的。如果出现错误，可以通过 http://groups.google.com/group/python-sqlite 上的活跃组报告给社区。

对于 Windows 平台，只需要下载和运行二进制安装文件。但一个前提条件是需要安装 SQLite3.2 或更新的版本。

连接

上文已经提到，Python2.5 及其之后的版本中包含的是 sqlite3 模块，或者使用单独源代码的 pysqlite 包 pysqlite2。DB API 模块在 dbapi2 中，当需要同时考虑 PySQLite 版本号和 SQLite 版本号时，这种情况有些混乱。通过模块的 connect()方法连接数据库，将需要打开的数据库的绝对或相对路径作为参数传递给该函数。

对于在 Python2.5+中包含的 pysqlite 而言，连接函数的实现如下：

```
import sqlite3

db = sqlite3.connect("foods.db")
```

对于独立编码的 pysqlite 而言，需要做如下改动：

```
from pysqlite2 import dbapi2 as sqlite3

db = sqlite3.connect("foods.db")
```

一般规则是：对于内存数据库，需要在数据库名中包含:memory:；创建新的数据库则要指定新的文件名，等等。从上面的例子中可以看出，两种连接数据库的方法只在 import 语句中不同。从现在开始，我们假设用户会选择最适合自己情况的 pysqlite。

查询处理

查询是根据 Python DB API Specification2 执行的，即创建一个游标对象，并执行查询，然后获得查询结果。在内部，一个游标对象含有一个 sqlite3_stmt 句柄。代码 8-6 解释了如何执行基本的查询。

228 代码 8-6　PySQLite 中的基本查询

```
from pysqlite2 import dbapi2 as sqlite3

# 连接
con = sqlite3.connect("foods.db")

# 准备一个语句并执行，这将调用 sqlite3_prepare()和 sqlite3_step()函数
cur = con.cursor()
cur.execute('select * from foods limit 10')

# 递归结果，打印字段名 (row[2])
row = cur.fetchone()
while row:
    print row[2]
    # 获取下一行
    row = cur.fetchone()

cur.close()
con.close()
```

注意，Python 会识别空格字符，所以，在复制代码时不要引入多余的空格或者其他空格字符。执行这段代码的结果如下：

```
Bagels
Bagels, raisin
Bavarian Cream Pie
Bear Claws
Black and White cookies
Bread (with nuts)
Butterfingers
Carrot Cake
Chips Ahoy Cookies
Chocolate Bobka
```

查询编译和执行的第一步都是在 execute()方法中完成的。这里，sqlite3_step()函数调用后，调用 sqlite3_prepare()函数。对那些只是修改数据而没有返回值的查询而言，到这里已经完成了查询。对于 select 语句，这里会返回结果集的第一行值。close()方法销毁语句句柄。

PySQLite 2 像其他 Python 数据库封装一样，支持递归遍历结果集。例如，代码 8-6

中的代码可以被重写为代码 8-7。

代码 8-7　在查询中使用 Pythonic 迭代器

```
from pysqlite2 import dbapi2 as sqlite3

con = sqlite3.connect("foods.db")
cur = con.cursor()
cur.execute('select * from foods limit 10')

for row in cur:
    print row[2]
```

229

参数绑定

PySQLite 支持按照位置或名字识别参数，这点也可以利用游标对象的 execute()方法实现，正如我们在代码 8-6 中所做的那样。编译、参数绑定和执行的第一步都在 execute()函数中完成。通过为 execute()函数的第二个参数传递一个数组，指定了一个位置参数。通过参数名传递了一个字典类型参数。代码 8-8 解释了这两种类型的参数绑定。

代码 8-8　PySQLite 中的参数绑定

```
from pysqlite2 import dbapi2 as sqlite3

con = sqlite3.connect("foods.db")
cur = con.cursor()
cur.execute('insert into episodes (name) values (?)', ('Soup Nazi'))
cur.close()

cur = con.cursor()
cur.execute('insert into episodes (name) values (:name)', {'name':'Soup Nazi'})
cur.close()
```

这个函数不支持重用编译过的查询（sqlite3_reset()）。要重用编译过的查询，可以使用函数 Cursor.executemany()。尽管 DB API 规定 executemany 应该接收一个列表类型作为参数，但是 PySQLite 扩展了参数的类型，使它适用于迭代器和产生器。例如，经典的使用方式如代码 8-9 所示。

代码 8-9　executemany()函数

```
from pysqlite2 import dbapi2 as sqlite3
```

```
con = sqlite3.connect("foods.db")
cur = con.cursor()

episodes = ['Soup Nazi', 'The Fusilli Jerry']
cur.executemany('insert into episodes (name) values (?)', episodes)
cur.close()
con.commit()
```

可以利用一个产生器更简单地实现上面的功能，如代码 8-10 所示。

230

代码 8-10　使用了产生器的 executemany()方法

```
from pysqlite2 import dbapi2 as sqlite3

con = sqlite3.connect("foods.db")
cur = con.cursor()

def episode_generator():
    episodes = ['Soup Nazi', 'The Fusilli Jerry']
    for episode in episodes:
        yield (episode,)

cur.executemany('INSERT INTO episodes (name) VALUES (?)', episode_generator())
cur.close()
con.commit()
```

在代码 8-9 和代码 8-10 中，查询的一次编译中使用了参数绑定技术。在代码和迭代器中重用查询，提高了对同一问题查询的性能。

PYSQLITE 中事务的处理

需要特别注意前面例子中的 con.commit()一行。如果使用 PySQLite，一定不要忘记 commit()函数。如果代码中没有调用该函数，那么例子中所有的事务都会被回滚。这个行为与 SQLite C API 以及其他扩展语言包不同。默认情况下，SQLite 运行在自动提交模式下。另外，PySQLite 在后台开始和终止一个事务，并根据使用的 SQL 类型，结合它自己的逻辑执行“开始”和“提交”事务。在向 SQLite 提交 SQL 语句前，PySQLite 会对它进行分析。如果发现 insert、update、delete 或者 replace 语句，PySQLite 就隐式地开始一个事务（发出 begin 指令）。之后，如果后面发出了其他类型的命令，PySQLite 在执行这些后面的命令前，会自动发出提交（commit）指令。PySQLite 的这种事务逻辑

限制了代码不能使用有冲突的回滚指令，否则，会干扰 PySQLite 对事务的监控。

可以禁止 PySQLite 的这种行为，恢复 SQLite 默认的行为。通过设置数据库连接（isolation_level）的值为 none，可以关闭 PySQLite 自动提交事务模式，如下所示：

```
from pysqlite2 import dbapi2 as sqlite3

# Turn on autocommit mode
con = sqlite3.connect("foods.db", isolation_level=None)
```

用户自定义函数

通过 create_function()函数可以注册用户自定义函数，其形式如下：

```
con.create_function(name, args, pyfunc)
```

231

这里，参数 name 指定 SQL 函数的函数名，参数 args 指定该函数接收参数的个数，参数 pyfync 是实现 SQL 功能的 Python 函数名。代码 8-11 显示了 Python 如何利用 PySQLite 实现 hello_newman()函数。

代码 8-11　PySQLite 中的 hello_newman()函数

```
from pysqlite2 import dbapi2 as sqlite3

# Python 函数
def hello_newman():
    return 'Hello Jerry'

con = sqlite3.connect(":memory:")
con.create_function("hello_newman", 0, hello_newman)
cur = con.cursor()

cur.execute("select hello_newman()")
print cur.fetchone()[0]
```

如果参数个数为-1，那么 hello_newman()函数将接受任意个数的参数。

聚合

PySQLite 将用户定义的聚合语句作为 Python 的类，这些类必须实现 step()和 finalize()

方法。使用 Connection::create_aggregate()函数注册聚合，与 create_function()函数一样，create_aggregate()函数也有三个参数，分别代表：函数的 SQL 名称、参数的个数，以及实现聚合的类名。代码 8-12 实现了一个称为 pysum 的简单的 SUM 聚合。

代码 8-12　PySQLite 中的 pysum()聚合语句

```
from pysqlite2 import dbapi2 as sqlite3

class pysum:
    def init(self):
        self.sum = 0

    def step(self, value):
        self.sum += value

    def finalize(self):
        return self.sum

con = sqlite3.connect("foods.db")
con.create_aggregate("pysum", 1, pysum)
cur = con.cursor()
cur.execute("select pysum(id) from foods")
print cur.fetchone()[0]
```

232

APSW（另一种 Python 接口）

APSW 是由 Roger Binns 编写和维护的。关于 APSW 的详细信息，可以在 http://code.google.com/p/apsw 中查到。尽管 APWS 和 pysqlite 的功能相似，语法风格也类似，但是 APSW 提供了范围更广的功能和 API。这里我们不重复介绍 APSW 中与 Python 相似的代码示例，只介绍 APSW 中对开发有重要帮助的功能，用户可以在需要时使用它们。

支持虚拟的表、VFS、blob I/O、备份和文件控制：APSW 提供了对 SQLite 最新功能的全面支持，以及 pysqlite 中不支持的一些数据库管理任务。

APSW Python Shell：APSW 包含了扩展的 Shell，该 Shell 可以用在使用 SQLite 命令行接口的地方，包含 CSV 格式化和处理、严格的错误处理，以及其他数据和信息格式化的选项。

事务一致性处理：APSW 总是按照您的期望处理事务，不像 pysqlite 那样，在您完

全不知情的情况下处理它们。

APSW 还有一些其他便于使用的功能，这些功能可以从 http://apidoc.apsw.googlecode.com/hg/pysqlite.html 中查阅。

Ruby

Ruby 扩展包是由 Jamis Buck 编写的，现在有一组开发人员在维护。您可以在 http://rubyforge.org/projects/sqlite-ruby 中查看 Ruby 扩展包的详细信息和文档。所有有关 SQLite3 扩展包的文档都可以在 http://sqlite-ruby.rubyforge.org/sqlite3/中查看，这些文档在编排上可能有些混乱。

安装

这个扩展包有两种安装方式：使用或者不使用 SWIG。Ruby 的配置文件会自动识别出系统中是否有 SWIG 存在。我们推荐使用 SWIG（C 扩展包）来编译扩展包，这种执行方式更加稳定。如果操作系统中包含包管理器，比如 Debian、Ubuntu 等，也可以使用这些包管理器进行安装。

本书写作时，sqlite-ruby 的最新版本是 1.3.1。从 http://rubyforge.org/projects/sqlite-ruby 处下载安装压缩包，解压安装包，运行以下三个命令：

```
fuzzy@linux $ tar xjvf sqlite3-ruby-1.3.1.tar.bz2
fuzzy@linux $ cd sqlite3-ruby-1.3.1
fuzzy@linux $ ruby setup.rb config
fuzzy@linux $ ruby setup.rb setup
fuzzy@linux $ ruby setup.rb install
```

233

如果已经安装了 Ruby gem，可以利用 Ruby 一步即可完成安装操作：

```
fuzzy@linux $ gem install --remote sqlite3-ruby
```

如果是从软件仓库安装，要保证仓库中所选的版本号与 Ruby 的版本号一致。因为旧的 SQLite 版本也支持 Ruby，请在平台上安装 libsqlite3-ruby 或者与之等价的版本。比如，在 Ubuntu 上：

```
fuzzy@linux $ ruby --version
ruby 1.8.6 (2007-09-24 patchlevel 111) [x86_64-linux]
```

```
fuzzy@linux $ sudo apt-get install libsqlite3-ruby libsqlite-ruby libsqlite-ruby1.8
... many apt messages fly past ...
```

连接

在加载 SQLite 扩展包前，首先必须导入 sqlite 模块，使用 load 或者 require 语句安装，命令如下：

```
require 'sqlite'
```

通过初始化一个 SQLite::Database 对象连接数据库，初始化时指定数据库文件的名字。默认情况下，通过列编号访问结果集，此外，还可以将 Database::results_as_hash 设置为 true，通过列名访问结果集。

```
require 'sqlite'
db = SQLite::Database.new("foods.db")
db.results_as_hash = true
```

查询处理

Ruby 的扩展包和 SQLite API 很相似，都提供预准备查询和封装好的查询。

预准备查询是通过 Database::prepare()函数生成的，这个函数回传一个 Statement 对象，该对象包含一个 sqlite3_stmt 结构。通过 Statement 的 execute 方法执行查询，产生一个 ResultSet 对象。可以传递一个数据块给 Statement，这种情况下产生的 ResultSet 对象保存到这个数据块中。如果不使用这个数据块，那么 Statement 将把 ResultSet 对象作为返回值返回，用 ResultSet 对象遍历返回结果的行。在内部使用 sqlite3_step()函数遍历结果集。有两种方法可以访问 ResultSet 中的行，其一是利用 each()迭代器遍历数据块，另外，可以利用 next()方法循环遍历，这种方法返回数组里的下一条记录或者当到达集合尾部时返回空值。代码 8-13 显示了 Ruby 中生成查询的过程。

234 **代码 8-13　在 Ruby 中生成预准备查询**

```
#!/usr/bin/env ruby

require 'sqlite3'

db = SQLite3::Database.new('foods.db')
```

```
stmt = db.prepare('select name from episodes')

stmt.execute do | result |
  result.each do | row |
    puts row[0]
  end
end

result = stmt.execute()
result.each do | row |
  puts row[0]
end

stmt.close()
```

完成查询后要调用 Statement.close()方法结束查询，这点十分重要。因为在内部一个 Statement 对象包含一个 sqlite3_stmt 结构，该结构使得查询可以重用，避免了重新编译查询。

参数绑定

Ruby 扩展包既支持位置绑定，也支持参数名绑定。我们可用函数 Statement::bind_param()和 Statement::bind_params()绑定参数，它的声明如下：

```
bind_param(param, value)
```

如果参数 param 的值是 Fixnum，代表按照位置（索引）绑定参数。否则，按照参数名绑定。bind_params()接收任意数量的参数，如果 bind_params()中的第一个参数是哈希列表，那么它将参数名映射为值。否则，它将参数列表中的每个参数看做是位置参数。代码 8-14 解释了这两种方式的参数绑定。

代码 8-14　Ruby 中的参数绑定

```
require 'sqlite3'

db = SQLite3::Database.new("foods.db")
db.results_as_hash = true

# 命令参数
stmt = db.prepare('select * from foods where name like :name')
```

235

```
stmt.bind_param(':name', '%Peach%')

stmt.execute() do |result|
  result.each do |row|
    puts row['name']
  end
end

# 位置参数

stmt = db.prepare('select * from foods where name like ? or type_id = ?')
stmt.bind_params('%Bobka%', 1)

stmt.execute() do |result|
  result.each do |row|
    puts row['name']
  end
end

# 释放读取锁
stmt.close()
```

如果查询中不需要使用参数，有一种处理查询的简便方法：使用 Database::query() 函数，这个函数中没有 Statement 对象，只返回一个 ResultSet 对象，比如代码 8-14 中的例子。

代码 8-14　在 Ruby 中使用 Database::query()函数

```
require 'sqlite3'

db = SQLite3::Database.new("foods.db")
db.results_as_hash = true

result = db.query('select * from foods limit 10')
result.each do |row|
  puts row['name']
end

result.reset()

while row = result.next
  puts row['name']
```

```
end
result.close()
```

与 Statement 对象一样，Result 对象也是一个轻量级的包含查询语句的包，表示编译好的查询。当调用 reset 函数时返回 Result 对象。在内部中，reset 调用 sqlite3_reset()函数，并且重新执行查询。与 Statement 对象不同的是，Result 对象不能与参数绑定。

其他一些函数我们这里就一笔带过了，查询函数包括 Database::get_first_row()，该函数返回查询的第一行结果；Database::get_first_value()函数返回查询的第一行第一列的结果。

用户自定义函数

236

用户自定义函数通过 Database::create_function()实现，它的声明如下：

```
create_function( name, args, text_rep=Constants::TextRep::ANY ) {|func, *args| ...}
```

这里，name 表示自定义函数的函数名，args 代表参数的个数（-1 代表参数个数不定），text_rep 对应 UTF 编码，可使用的编码包括 UTF8、UTF16LE、UTF16BE、UTF16 和 ANY。最后，花括号中定义了实现对应功能的函数。代码 8-15 诠释了在 Ruby 中如何实现自定义函数 hello_newman()。

代码 8-15　Ruby 中的 hello_newman()函数

```
require 'sqlite3'

db = SQLite3::Database.new(':memory:')

db.create_function('hello_newman', -1 ) do |func, *args|
  if args.length == 0
     func.result = 'Hello Jerry'
  else
     func.result = 'Hello %s' % [args.join(', ')]
  end
end

puts db.get_first_value("SELECT hello_newman()")
puts db.get_first_value("SELECT hello_newman('Elaine')")
puts db.get_first_value("SELECT hello_newman('Elaine', 'Jerry')")
puts db.get_first_value("SELECT hello_newman('Elaine', 'Jerry', 'George')")
```

运行上面的代码，产生如下结果：

```
Hello Jerry
Hello Elaine
Hello Elaine, Jerry
Hello Elaine, Jerry, George
```

Java

像前面介绍的 Ruby 一样，Java 也有很多支持 SQLite 的扩展包，这些扩展包大致分为两类：JDBC 驱动和 Java 中对 SQLite 完全重新的实现。这里我们将介绍由 Christian Werner 编写的 javasqlite 封装，他还编写了 SQLite ODBC 驱动器。这个封装包括 JDBC 驱动和 JNI 扩展包，它们都与 SQLite C API 很相似。您可以从 www.ch-werner.de/javasqlite/overview-summary.html#jdbc_driver 中下载这些扩展包。

JNI 扩展包中的主要类是 SQLiteDatabase 类，该类的大部分方法都使用了回调机制，主要实现了下面的接口：

- SQLite.Callback
- SQLite.Function
- SQLite.Authorizer
- SQLite.Trace
- SQLite.ProgressHandler

SQLite.Callback 接口通过行句柄、列句柄以及类型信息处理结果集合。SQLite.Authorizer 是包含了 SQLite C API 中的 sqlite3_set_authorizer()方法的轻量级封装，这个接口可以在数据库事务发生前截获它们。SQLite.Trace 包括了 sqlite3_trace()函数，可以用来在 SQL 语句编译时查看 SQL 语句。SQLite.ProgressHandler 包括了 sqlite3_progress_handler()函数，当处理了一定数量的 VDBE 指令后，该函数将给出事务的进展。

安装

当前版本的 SQLite 要求 JDK 1.4 或更高版本 JDK。Java 扩展包使用 GNU 自动配置、编译和安装，仅需要三步：

```
fuzzy@linux $ tar xvzf javasqlite-20100727.tar.gz
```

```
fuzzy@linux $ ./configure
fuzzy@linux $ make
fuzzy@linux $ make install
```

配置脚本会在几个默认位置查找 SQLite 和 JDK。或者可以使用如下几个命令行指定 SQLite 和 JDK 的位置。

```
--with-sqlite=DIR  指定 sqlite v2 所在的路径
--with-sqlite3=DIR  指定 sqlite v3 所在的路径
--with-jdk=DIR  指定 JDK 所在的路径
```

利用-prefix=DIR 选项，指定结果库（libsqlite_jni.so 文件）的安装路径，默认路径是/usr/lib/jvm/java-6-sun/jre/lib/i386（这个路径会因 Linux 版本，或者是其他 UNIX、Mac OS X 平台的不同而不同，比如有的可能是/usr/local/lib）。利用--withjardir=DIR 选项，指定 sqlite.jar 的安装路径，默认路径是/usr/lib/jvm/java-6-sun-1.6.0.20/jre/lib/ext/sqlite.jar（同样，这个路径也会因为操作系统类型和版本的不同而不同）。这个 jar 文件包含上层的部分和 JDBC 驱动，运行时，需要使用-classpath 和-Djava.library.path=..命令行参数通知 JVM 这两者的路径。

在 Windows 中，扩展包中包含构建稳健的 javasqlite.mak 和 javasqlite3.mak。这两个文件包含一些编译指令，使用 Sun 提供的 J2SE 1.4.2 和 Microsoft Visual C++ 6.0。JNI 部分包含 SQLite 3.7.0 的 DDL 和 JAR 文件的可以从网站上下载。

连接

238

使用 SQLiteDatabase::open()函数可以连接数据库，该函数包含数据库文件名和文件类型参数，如代码 8-16 所示。文件类型参数是 SQLite 2 API 中定义的，现在已经没有任何意义了，可以为这个参数赋任何值。代码 8-16 中的代码节选自 SQLiteJNIExample.java 文件，这段代码解释了几乎所有数据库常用的操作：连接、查询、函数的使用等。

代码 8-16　JavaSQLite 测试代码

```
import SQLite.*;

public class SQLiteJNIExample
{
  public static void main(String args[])
  {
```

```
SQLite.Database db = new SQLite.Database();

try
  {
    db.open("foods.db", 700);

    // 追踪 SQL 语句
    db.trace(new SQLTrace());

    // 查询示例
    query(db);

    // Function example
    user_defined_function(db);

    // 聚合示例
    user_defined_aggregate(db);

    db.close();
  }
  catch (java.lang.Exception e)
    {
      System.err.println("error: " + e);
    }
  }

  ...
}
```

查询处理

函数 SQLiteDatabase::compile()可以对查询进行编译，该函数可以处理包含多个 SQL 语句的字符串，返回一个包含所有查询语句的 VM（虚拟机）对象。

239 每次调用 compile()函数时，VM 对象会解析一条 SQL 语句，如果查询编译成功，则返回 true，否则返回 false。借助这个方法可以循环遍历所有的 SQL 语句，当处理完最后一条 SQL 语句后中止。

当一条查询语句被 VM 编译后，可以利用 VM::step()函数执行它。VM::step()函数需要一个实现了 SQLite.Callback 接口的对象作为该参数，代码 8-17 的例子中一个名为

Row 的类就是这个参数。

代码 8-17　Row 类

```
class Row implements SQLite.Callback
{

  private String row[];

  public void columns(String col[]) {}
  public void types(String types[]) {}

  public boolean newrow(String data[])
  {
    // 复制到内部数据
    row = data;
    return false;
  }

  public String print()
  {
    return "Row:    [" + StringUtil.join(row, ", ") + "]";
  }
}
```

SQLite.Callback 接口有三个方法：columns()、types()和 newrow()，它们分别处理列名、列的类型和行数据。每个函数都调用 VM::step()函数更新列、类型和行信息。

代码 8-18 的例子解释了 query()函数中 VM 的用法。

代码 8-18　query()函数

```
public static void query(SQLite.Database db)
  throws SQLite.Exception
{
  System.out.println("\nQuery Processing:\n");

  Row row = new Row();
  db.set_authorizer(new AuthorizeFilter());

  Vm vm = db.compile( "select * from foods limit 5;" +
                      "delete from foods where id = 5;" +
                      "insert into foods (type_id, name) values (5, 'Java');" +
```

```
                    "select * from foods limit 5" );
  do
  {
240   while (vm.step(row))
    {
      System.err.println(row.print());
    }
  }
  while (vm.compile());
}
```

SQLite.Database::exec()函数执行自包含的查询，其函数声明如下：

```
void exec(String sql, Callback cb, String[] params)
```

params 数组对应 SQL 语句中的%q 或者%Q 参数。代码 8-19 展示了这一示例。

代码 8-19　exec_query()函数

```
public static void exec_query(SQLite.Database db)
  throws SQLite.Exception
{
  System.out.println("\nExec Query:\n");
  String sql = "insert into foods (type_id, name) values (5, '%q')";
  ResultSet result = new ResultSet();

  String params[] = {"Java"};
  db.exec(sql, result, params);

  System.out.println("Result: last_insert_id(): " + db.last_insert_rowid());
  System.out.println("Result:         changes(): " + db.changes());
}
```

■注意：这里与参数绑定不同。相反，这里使用 SQLite C API 中 sprintf 格式替代者 sqlite3 Vmprintf()。

自定义函数和聚合

SQLite.Function 接口用于实现用户自定义函数和聚合。代码 8-20 显示了 Java 扩展包中的自定义函数 hello_newman()。

代码 8-20　Java 中的 hello_newman()函数

```
class HelloNewman implements SQLite.Function
{
  public void function(FunctionContext fc, String args[])
  {
    if (args.length > 0)
    {
      fc.set_result("Hello " + StringUtil.join(args, ", "));
    }
    else
    {
      fc.set_result("Hello Jerry");
    }
  }

  public void step(FunctionContext fc, String args[]){}
  public void last_step(FunctionContext fc)
  {
    fc.set_result(0);
  }
}
```


注意最后的 step()和 last_step()两个函数，尽管它们仅仅是定义聚合时才用到，与用户自定义函数无关，但还是要在函数体中实现它们，这是因为 SQLite.Function 接口定义的函数都是同时针对函数和聚合的。例子中的类通过 SQLite.Database.create_function()函数注册，该函数的返回类型还需要通过 SQLite.Database.function_type()函数注册。代码 8-21 显示了实现 hello_newman()函数的 Java 代码。

代码 8-21　hello_newman()测试代码

```
// 注册函数
db.create_function("hello_newman", -1, new HelloNewman());

// 设置返回类型
db.function_type("hello_newman", Constants.SQLITE_TEXT);

// 测试
PrintResult r = new PrintResult();
db.exec("select hello_newman()", r);
db.exec("select hello_newman('Elaine', 'Jerry')", r);
db.exec("select hello_newman('Elaine', 'Jerry', 'George')", r);
```

JDBC

Java 扩展包还包括对 JDBC 的支持。若要使用 JDBC 驱动，需要将 JDBC 驱动的类名指定为 SQLite.JDBCDriver。同样，确保 sqlite.jar 在类路径中，本地类库在 Java 类库路径中。JDBC URL 连接 SQLite 数据库的格式为：jdbc:sqlite:/path，其中，path 指定了 SQLite 数据库所在的路径，例如：

```
jdbc:sqlite://dirA/dirB/dbfile
jdbc:sqlite:/DRIVE:/dirA/dirB/dbfile
jdbc:sqlite:///COMPUTERNAME/shareA/dirB/dbfile
```

目前，SQLite 数据库表支持的数据类型包括：java.lang.String、short、int、float 和 double，还有一些支持 java.sql.Date、java.sql.Time 和 java.sql.Timestamp。连接属性 daterepr 定义这些数据类型的行为。如果 daterepr=Julian，那么 SQLite 在这些数据类型上进行的所有插入/更新操作解释为浮点数转换为等价的 Julian 日期。其他一些值的转换行为包括字符串 YYYY-mm-dd 转换为日期，HH:MM:SS 转换为时间，YYYY-mm-dd HH:MM:SS.f 转换为时间戳。

242

其他数据类型的转换依赖于 SQLite 中对编译指示 show_datatypes 和 table_info 的设置。扩展包里实现了很多有关基本数据库元数据的方法，因此，可以使用 JDK 1.4 或者更新版本的 JDK 和 iSQL-Viewer 工具访问 SQLite 数据库。

代码 8-22 显示了一个使用 JDBC 驱动查询 foods 数据库的代码，这段代码可以在 SQLiteJDBCExample.java 中找到。

代码 8-22　SQLite JDBC 测试程序

```
import java.sql.*;
import SQLite.JDBCDriver;

public class SQLiteJDBCExample {

public static void main ( String [ ] args )
{
  try
  {
    Class.forName("SQLite.JDBCDriver");
    Connection c = DriverManager.getConnection( "jdbc:sqlite://tmp/foods.db",
                                                "" // username (NA),
                                                "" // password (NA));
```

```
  Statement s = c.createStatement();
  ResultSet rs = s.executeQuery ("select * from foods limit 10");
  int cols = (rs.getMetaData()).getColumnCount();

  while (rs.next())
  {
    String fields[] = new String[cols];

    for(int i=0; i<cols; i++)
    {
      fields[i] = rs.getString(i+1);
    }

    System.out.println("[" + join(fields, ", ") + "]");
  }
}
catch( Exception x )
{
  x.printStackTrace();
}
}

static String join( String[] array, String delim )
{
  StringBuffer sb = join(array, delim, new StringBuffer());
  return sb.toString();
}

static StringBuffer join( String[] array, String delim, StringBuffer sb )
{
  for ( int i=0; i<array.length; i++ )
  {
    if (i!=0) sb.append(delim);
    sb.append("'" + array[i] + "'");
  }

  return sb;
}
}
```

243

运行上述代码的输出结果是：

```
['1',   '1',   'Bagels']
['2',   '1',   'Bagels, raisin']
['3',   '1',   'Bavarian Cream Pie']
['4',   '1',   'Bear Claws']
['6',   '1',   'Bread (with nuts)']
['7',   '1',   'Butterfingers']
['8',   '1',   'Carrot Cake']
['9',   '1',   'Chips Ahoy Cookies']
['10',   '1',   'Chocolate Bobka']
['11',   '1',   'Chocolate Eclairs']
```

Tcl

SQLite 的开发者编写了 Tcl 扩展包，并在进行维护。所有的 SQLite 测试代码都已经在 Tcl 里测试通过，测试使用的是 SQLite Tcl 扩展包。可以负责地说，Tcl 扩展包是稳定的，而且是经过充分测试的。Tcl 扩展包已经成为 SQLite 源代码的一部分，您可以在 SQLite 的网站（www.sqlite.org/tclsqlite.html）找到 Tcl 扩展包的完整文档。

安装

SQLite GNU Autoconf 脚本会自动搜索 Tcl，如果找到，则自动编译生成 Tcl 扩展包。推荐使用 SQLite 网站上提供的针对 Linux 和 Windows 的压缩包或者预编译的 Tcl 扩展包编译安装 Tcl 接口。当编译 Tcl 扩展包，并安装到默认路径时，会看到自动运行“编译和生成”的过程，所以，Tcl 和 Tcl/Tk 脚本会正确加载扩展包。您可以在 SQLite 包的
244 目录下找到必需的库，比如，在 Debian 或者 Ubuntu 下，可以使用下面的命令添加 Tcl 扩展包：

```
fuzzy@linux $ sudo apt-get install libsqlite3-tcl
```

在撰写本章时，Tcl 8.6 版本中默认包含了 SQLite，这样安装过程就更加简单了。

连接

SQLite 扩展包位于 sqlite3 包中，必须使用指令 package require 加载它。使用 sqlite3 命令创建一个数据库句柄，连接数据库。这个命令包括两个参数：第一个参数指定要创

建的数据库句柄的名字，第二个参数指定数据库文件的路径。下面是一个说明连接数据库的例子：

```
#!/usr/bin/env tclsh

package require sqlite3

puts "\nConnecting."
sqlite3 db ./foods.db
```

这里还是遵循通用的数据库连接规则。如果用:memory:取代数据库文件名，将会创建一个内存数据库。如果第二个参数指定的数据库文件不存在，则会创建一个新的数据库文件。返回的数据库句柄对应连接到指定的数据库，但是这个数据库只有在真正开始使用它时，才会被打开。这个数据库句柄是唯一一个与数据库连接的对象，贯穿于数据库使用的始终。

若不需要连接时，可使用数据库句柄的 close 方法断开与数据库的连接。断开时，会自动回滚被挂起的事务。

查询处理

扩展包使用 eval 方法执行查询，该方法可以一次处理一个或者多个查询。eval 有多种使用方法，第一种方法是按照 SQL 代码的顺序遍历脚本中所有的查询。脚本每执行一次，都会在结果集中返回一行结果，脚本中每一行的域都是局部变量。这里有一个例子：

```
puts "\nSelecting 5 records."
db eval {select * from foods limit 5} {
    puts "$id $name"
}
```

另一种形式是将返回结果中多个域的值赋值给数组。这时，需要在 SQL 语句的后面、脚本的前面指定数组名。其示例如下：

```
puts "\nSelecting 5 records."
db eval {select * from foods limit 5} values {
    puts "$values(id) $values(name)"
}
```

245 运行上述两个例子将产生如下结果：

```
Selecting 5 records.

1   Bagels
2   Bagels, raisin
3   Bavarian Cream Pie
4   Bear Claws
5   Black and White cookies
```

如果代码中不提供花括号里的脚本，那么 eval 将返回结果集，该结果集是一个值很长的代码，由程序员决定记录间的边界。可以参考下面的例子：

```
set x [db eval {select * from foods limit 3}]
```

这条 SQL 语句将返回如下值：

```
{1 1 Bagels 2 1 {Bagles, raisin} 3 1 {Bavarian Cream Pie}}
```

这个返回结果对应三条记录，每条记录有三个域（id、type_id 和 name）。

对于非 select 语句，eval 将返回已修改记录的信息，比如代码 8-23 中的例子。

代码 8-23 在 Tcl 中检查数据库修改记录

```
db eval begin

puts "\nUpdating all rows."
db eval { update foods set type_id=0 }
puts "Changes             : [db changes]"

puts "\nDeleting all rows."
# 删除所有的行
db eval { delete FROM foods }

puts "\nInserting a row."
# 插入一行
db eval { insert into foods (type_id, name) values (9, 'Junior Mints') }

puts "Changes             : [db changes]"
puts "last_insert_rowid() : [db last_insert_rowid]"

puts "\nRolling back transaction."
db eval rollback
```

运行代码 8-23 将产生下列结果：


```
Updating all rows.
Changes                : 415

Deleting all rows.

Inserting a row.
Changes                : 1
last_insert_rowid()    : 1

Rolling back transaction.
Total records          : 415
```

在 Tcl 代码中可以通过 transaction 方法自动处理事务的范围。如果 transaction 方法里所有的脚本语句都能正确执行完毕，那么 transaction 方法将提交事务；反之，它会回滚事务。比如，在单个事务中执行 8-23 的代码，在每条命令执行完毕后要检查每条命令是否执行成功。如果执行失败了，那么需要回滚事务，放弃后面的所有操作。在事务中执行的命令越多，代码会越难管理。但是，通过 transaction 方法自动管理事务，会降低管理难度，比如代码 8-24 中的例子。

代码 8-24　Tcl 中的事务范围

```
db transaction {
puts "\nUpdating all rows."
db eval { update foods set type_id=0 }
puts "Changes                : [db changes]"

puts "\nDeleting all rows."

# 删除所有行
db eval { delete from foods }

puts "\nInserting a row."

# 插入一行
db eval { insert into foods (type_id, name) values (9, 'Junior Mints') }

puts "Changes                : [db changes]"
puts "last_insert_rowid()    : [db last_insert_rowid]"
}
```

现在，任意一条命令失败后，transaction 都会回滚所有的命令操作，不需要检查任何返回值代码。Transaction 对于已经存在的事务也是有效的，只要事务已经开始了，transaction 方法对它就是有效的。对于已经开始的事务，transaction 会检查命令是否执行成功，但是不会试图提交或者回滚。如果有错误发生，transaction 会终止脚本，返回适当的错误代码。

247 用户自定义函数

利用 function 函数可以创建用户自定义的函数，创建时要指定函数名和函数体。代码 8-25 给出了一个在 Tcl 中定义的 hello_newman()函数。

代码 8-25　Tcl 中的 hello_newman()函数

```
proc hello_newman {args} {
    set l [llength $args]
    if {$l == 0} {
        return "Hello Jerry"
    } else {
        return "Hello [join $args {, } ]"
    }
}

db function hello_newman hello_newman
puts [db onecolumn {select hello_newman()}]
puts [db onecolumn {select hello_newman('Elaine')}]
puts [db onecolumn {select hello_newman('Elaine', 'Jerry')}]
puts [db onecolumn {select hello_newman('Elaine', 'Jerry', 'George')}]
```

运行代码 8-25 会产生如下结果：

```
Hello  Jerry
Hello  Elaine
Hello  Elaine,  Jerry
Hello  Elaine,  Jerry,  George
```

PHP

PHP 5 发行前，SQLite 已经成为 PHP 标准库的组成部分。如果使用 PHP 5 或者之

后的版本，那么 SQLite 已经安装到了系统中。与之前介绍的几种扩展语言包一样，PHP 也提供了大量的接口选项。

PHP 5.1 引进了一种新的数据库抽象层——PHP Data Objects（PDO），它已经成为 PHP 最常使用的数据库抽象层之一。这个 API 使用驱动支持一个标准的数据库接口。PDO 接口是一种面向对象（OO）的接口，与 SQLite 扩展包里的 OO 接口很相似。因为 PDO 是一个抽象层，因此，它适用于很多不同的数据库，具有普遍性。尽管如此，PDO 驱动还是提供了一些针对不同数据库的特殊功能。也就是说，PDO 驱动提供了一个完整的 OO 接口，适用于 SQLite 的所有功能。

安装

248

PHP 的安装很简单，但不在本书的讨论范围。您可以在网站 www.php.net 中找到编译和安装 PHP 的详尽说明文档。

连接

在打开一个数据库之前，需要考虑两个问题：位置和权限。默认情况下，PHP 中的路径是相对路径，即相对于当前脚本所在的位置（除非提供一个绝对路径）。所以，如果只是指定了一个数据库名，那么将会在局域网的公共位置打开或者创建一个数据库，数据库的权限取决于网络服务器的配置。

因为 SQLite 数据库文件和 HTML 文件、图像文件一样是普通的操作系统文件，所以可以像普通文件那样获得它们。这就导致了潜在的安全问题，安全与否取决于数据的敏感度。一个较好的安全预防办法是把数据库文件保存在公共文件夹以外，只允许 PHP 脚本访问；或者可以采用更复杂的安全措施，利用.htaccess 控制或者等价的安全技术控制对 SQLite 数据库文件的访问。

无论最终使用哪种权限，都要保证网络服务器允许运行的 PHP 读写数据库文件。这些都是管理权限的细节，需要针对不同的情况做不同的处理。

在 PDO 中，连接封装在 PDO 类中，该类的构造函数只有一个参数，即数据来源名（DSN）。DSN 是一个由两个参数组成的字符串，这两个参数间用冒号分隔。第一个参数是驱动名，这个值通常是 sqlite，对应 SQLite 3。第二个参数是数据库文件的路径。如果连接因为某种原因失败了，那么构造函数会抛出一个 PDOException。下面的例子连

接到了 foods.db 数据库。

```
<?php
try {
  $dbh = new PDO("sqlite:foods.db");
} catch (PDOException $e) {
  echo 'Connection failed: ' . $e->getMessage();
}?>
```

查询

PDO 由两个基本类组成——PDO 和 PDOStatement。PDO 表示一个连接，在内部包含一个 sqlite3 结构体，PDOStatement 表示一个语句句柄，内部包含一个 sqlite3_stmt 结构体。PDO 类中的查询方法和 SQLite C API 中的同类方法很相似。有三种方法可以执行查询。

- **exec()**：执行不返回数据的查询。返回语句影响行的数量，或者发生错误时返回 FALSE。这个函数与 sqlite3_exec()机制相同。
- **query()**：执行查询并返回一个表示结果集的 PDOStatement 对象，或者发生错误时返回 FALSE。
- **prepare()**：编译查询并返回一个 PDOStatement 对象，或者遇到错误时返回 FALSE。这个函数的性能比 query()函数好，与 query 函数不同的是，prepare 函数在进行查询时不需要对同一个查询编译多次，它可以重置，避免了重新编译查询。

249

另外，PDO 中的事务管理可以通过调用 beginTransaction()、commit()和 rollback()函数完成。代码 8-26 给出了一个利用 PDO 类打开数据库并在事务中执行查询的例子。

代码 8-26　PDO 的基本查询

```
<?php

try {
  $dbh = new PDO("sqlite:foods.db");
} catch (PDOException $e) {
  echo 'Connection failed: ' . $e->getMessage();
}

$dbh->beginTransaction();
$sql = 'select * from foods limit 10';
```

```
foreach ($dbh->query($sql) as $row) {
  print $row['type_id'] . " ";
  print $row['name'] . "<br>";
}

$dressing = $dbh->quote("Newman's Own Vinegarette");
$dbh->exec("insert into foods values (NULL, 4, $dressing)");
echo $dbh->lastInsertId();
$dbh->rollback();
?>
```

PDO 通过 setAttribute()和 getAttribute()函数设置数据库连接中的不同参数。这些参数中只有 PDO_ATTR_TIMEOUT 参数是用于 SQLite 的，该参数用来设置超时时间。

函数 prepare()使用一条 SQLite 语句对结果集定位，它同时支持按名称或者位置查找参数。PHP 中的变量必须与 SQL 的参数绑定，PHP 变量与 SQL 参数间的绑定通过 PDOStatement::bindParam()函数完成，该函数声明如下：

```
nool bindParam ( mixed parameter, mixed &variable,
                 int data_type, int length,
                 mixed    driver_options );
```

参数 paparm 指定 SQL 的参数。对于按位置查找参数的 SQL 语句而言，它是整数值。对于按参数名查找的参数，这个值是字符串。参数 variable 是 PHP 中与 SQL 参数绑定的变量名。data_type 指定变量类型。最后，如果 variable 是字符串，参数 length 是字符串变量的长度，表示返回字符串的最大长度。代码 8-27 给出了一个按照位置查找参数的例子。

代码 8-27　PDO 中按位置查找的参数

250

```
<?php
$dbh = new PDO("sqlite:foods.db");
$sql = 'select * from foods where type_id=? And name=?';
$stmt = $dbh->prepare($sql);
$type_id = 9;
$name = 'JujyFruit';
$stmt->bindParam(1, $type_id, PDO_PARAM_INT);
$stmt->bindParam(1, $name, PDO_PARAM_STR, 50);
$stmt->execute();
?>
```

参数名绑定机制与参数位置绑定机制相似。按照参数名绑定时，要指定 SQL 语句中的参数名（而不是位置）。代码 8-28 将代码 8-27 中的例子用参数名绑定方法进行了修改。

代码 8-28　PDO 中的参数名绑定

```
<?php
$dbh = new PDO("sqlite:foods.db");
$sql = 'select * from foods where type_id=:type And name=:name;';
$stmt = $dbh->prepare($sql);
$type_id = 9;
$name = 'JujyFruit';
$stmt->bindParam('type', $type_id, PDO_PARAM_INT);
$stmt->bindParam('name', $name, PDO_PARAM_STR, 50);
$stmt->execute();
?>
```

PDO 还可以将结果集中的列赋值给 PHP 中的变量，通过 PDOStatement::bindColumn() 函数完成，该函数声明如下：

```
bool bindColumn (mixed column, mixed &param, int type)
```

参数 column 指定列名或者 SELECT 语句的索引。param 对应 PHP 中的变量名，参数 type 指定 PHP 变量的类型。下面的例子是将变量$name 和$type_id 与结果集绑定。

```
<?php
$dbh = new PDO("sqlite:foods.db");
$sql = 'select * from foods limit 10';
$stmt = $dbh->prepare($sql);
$stmt->execute();
$name;
$type_id;
$stmt->bindColumn('type_id', $type_id, PDO_PARAM_INT);
$stmt->bindColumn('name', $name, PDO_PARAM_STR);
while ($row = $stmt->fetch()) {
  print "$type_id $name <br>";
}
```

用户自定义函数和聚合

用户自定义函数通过函数 sqliteCreateFunction()实现。用户自定义聚合通过函数

sqliteCreateAggregate()实现。函数 sqliteCreateFunction()声明如下：

```
void PDO::createFunction ( string function_name,
                           callback callback,
                           int num_args );
```

其中参数的含义如下。

- function_name：自定义函数在 SQL 中的函数名称。
- callback：当 SQL 函数被调用时，被调用的 PHP（回调）函数。
- num_args：回调函数的参数个数。

下面是在 PHP 中实现的 hello_newman()函数的例子：

```
<?php
function hello_newman() {
    return 'Hello Jerry';
}

$db = new PDO("sqlite:foods.db");
$db->createFunction('hello_newman', hello_newman, 0);
$row = $db->query('SELECT hello_newman()')->fetch();
print $row[0]
?>
```

可以用类似的方法使用 sqliteCreateAggregate 创建聚合，函数 sqliteCreateAggregate 的声明如下：

```
void PDO::createAggregate ( string function_name,
                            callback step_func,
                            callback finalize_func,
                            int num_args )
```

下面的代码实现了之前定义的 SUM 聚合，称为 phpsum：

```
<?php
function phpsum_step(&$context, $value) {
    $context = $context + $value;
}

function phpsum_finalize(&$context) {
    return $context;
}
```

252

```
$db = new PDO("sqlite:foods.db");
$db->createAggregate('phpsum', phpsum_step, phpsum_finalize);
$row = $db->query('select phpsum(id) from food_types')->fetch();
print $row[0]
?>
```

总结

本章简要介绍了几种不同的语言扩展包，以及它们如何与 SQLite 工作。尽管使用 SQLite 中的 C API 比较直接，但是使用 SQLite 语言扩展包更加方便。这两者在许多方面很相似，许多扩展语言包提供的函数都可以在 SQLite C API 中找到与之对应的方法。尽管不同语言的数据库的接口有所不同，但是它们在本质是一致的。所有的查询最终都包含一个数据库连接对象，这个对象映射到内部的一个 sqlite3 结构体、一个查询语句或一个游标对象，该语句或游标对象在内部映射到一个 sqlite3_stmt 结构体。

这些扩展包使得 SQLite 的使用更加方便和简单，增加了 SQLite 的可用性，使得 sqlite 可以应用在从系统管理到网络开发的多种程序中。近年来，扩展语言包的数量急剧增加，在 SQLite wiki 网站上（www.sqlite.org/cvstrac/wiki?p=SqliteWrappers）可以查找这些扩展包。

第 9 章

iOS 开发中的 SQLite

您可能是一名资深的 SQLite 用户或者开发者，也可能阅读这本书是您第一次走进 SQLite 世界。无论您具有何种背景，每天都在不知不觉中使用着 SQLite。因为 SQLite 是一款被广泛应用于移动设备的数据库，从 iPhone、iPod 到最新的平板电脑、iPad、媒体播放器等。

本章中，我们将从另一个角度介绍如何使用 SQLite 进行开发。重点探讨苹果公司的移动设备对 SQLite 的实现和支持。我们之前或许提到过将它称为“iPhone 开发”，苹果公司将其移动设备上的操作系统统一命名为 iOS。

在第 5 章中介绍过如何将一个 API 映射为另一个 API，您可能对此印象不深，因为，目前为止，您只是刚刚学习 SQLite 包的很多内容，并未深入理解和掌握。本章我们将换一种讲解方式：在全面介绍 iOS SQLite 技术的同时，将重点放在搭建一个真实的可以工作的基于 SQLite 的 iOS 应用程序。当拿着自己开发的 Seinfeld 食品零售店 iOS 应用软件炫耀时，您将成为周围人的羡慕对象。我们将这款软件称为 iSeinfeld...真是个再好不过的名字。

另外，我们还将介绍一些开发过程中必须要考虑的因素，这些因素对成功地在一台设备上长时间维护一个 iOS 应用至关重要。

SQLite iOS 开发的先决条件

在 iOS 上使用 SQLite 进行开发之前，需要安装一些必要的工具。严格地讲，这些工具不只是针对 SQLite 开发的。安装好这些工具后，就搭建好了一个完整的 iOS 开发环境，可以针对 iPhone、iPod 和 iPad 进行基于 SQLite 或者其他基于 Objective-C 的应用

程序。

搭建 iOS 开发环境可以笼统地概括为以下三步：

- 注册成为 Apple 软件开发者。
- 下载并安装 Xcode 和 Apple IDE。
- 下载并安装 iOS SDK。

在介绍 SQLite 开发前，我们先简单介绍以上三步。

254 注册成为 Apple 软件开发者

如果想为 iPad、iPhone 或者任何苹果设备开发并发布应用程序，需要注册为 Apple 的软件开发者。最初，我们把注册步骤一带而过，让读者自己为他们的设备进行注册。但是，后来我们发现，注册过程中细微的差别可能阻碍许多对 iOS 开发感兴趣的开发者。因此，介绍它是有必要的。

访问 Apple 开发者网站就会发现，需要花费 99 美元注册成为 iOS 开发者计划会员。这个信息会很明显地出现在 iOS 开发者网站的主页上（http://developer.apple.com/programs/ios）。虽然只有支付费用成为会员才能通过 iTunes Store 发布应用程序（这样可以省去很多申请核准的步骤），但是只需要注册成为 Apple 开发者就能够免费得到 iOS 开发所需要的工具包、SDK 等。

简单注册成为开发者是免费的！刚开始访问 Apple 网站的网页时可能有点找不到头绪，但是如果按照本书提示，直接访问 http://developer.apple.com/programs/register，您将很顺利地注册成功。

当然，如果您逐渐喜欢上了编写基于 SQLite 的 iOS 应用程序，那么可以通过 iTunes 随时升级您的注册信息以满足发布应用程序的要求。

下载并安装 Xcode 和 iOS SDK

注册成为 Apple 开发者后，下一步需要下载 Apple 的集成开发环境——Xcode 和 iOS SDK。iOS SDK 包含 Xcode，也就是说，没有必要分别下载 Xcode 和单独的 SDK。iOS SDK 也包含绑定版本的 SQLite（这点我们随后将会讨论），这就避免了下载和维护单独版本的 SQLite 库。但是，这种捆绑方式会让第一次使用 Apple iOS 的开发者手足无措，因为人们会本能地下载 Xcode 作为集成开发环境，然后将它们加入 iOS SDK 中。如果

采用这种方法，您会发现还需要额外下载很多没有必要的软件。

下载 iOS SDK（包含 Xcode）更直接。首先需要用您的 Apple 开发者账号登录，然后单击首页上的“Download”链接进入下载页面，选择 Xcode 3.2.4 和 iOS SDK 3.1。这样就下载了一个名为 xcode_3.2.4_and_ios_sdk_4.1.dmg、大小约为 3.3GB 的文件。

■**注意**：当前 3.2.x 版本的 Xcode 和 4.x 版本的 iOS SDK 只能运行在 Snow Leopard 或者更新版本的 OS X 上。苹果公司为那些依然运行 Leopard（10.5.x 版本）或者更老版本的用户提供了版本较低的可兼容的 Xcode 和 iPhone SDK（苹果公司将其称为 iPhone SDK，而不是 iOS SDK）。您需要自己寻找该版本，本书不介绍该版本。如果在苹果网站上找不到，建议使用 Google 搜索苹果论坛，该论坛上很多热心的开发者还保留着低版本的 SDK 文件夹.dmg。

下载完 iOS SDK 后，就可以开始安装了。打开.dmg 文件会看到三个文件项，如图 9-1 所示，这三个文件分别为：描述 SDK 的 About 文件（About iPhone SDK）、SDK 的安装文件（iPhone SDK）和一个名为 Packages 的文件夹。 255



图 9-1　iOS SDK 安装包内容

现在运行 iOS SDK 安装文件（或者 iPhone SDK 安装文件，这取决于所使用的版本）。硬盘运行几分钟后，将会出现如图 9-2 所示的安装欢迎界面。

除了以上欢迎界面外，这里没有必要把安装过程中所有的步骤一一呈现出来。在安装的每一步中单击“Continue”按钮，选择默认设置，或者根据需要修改一些选项。这 256

里值得一提的是，安装过程中选择安装类型这步，如图 9-3 所示。

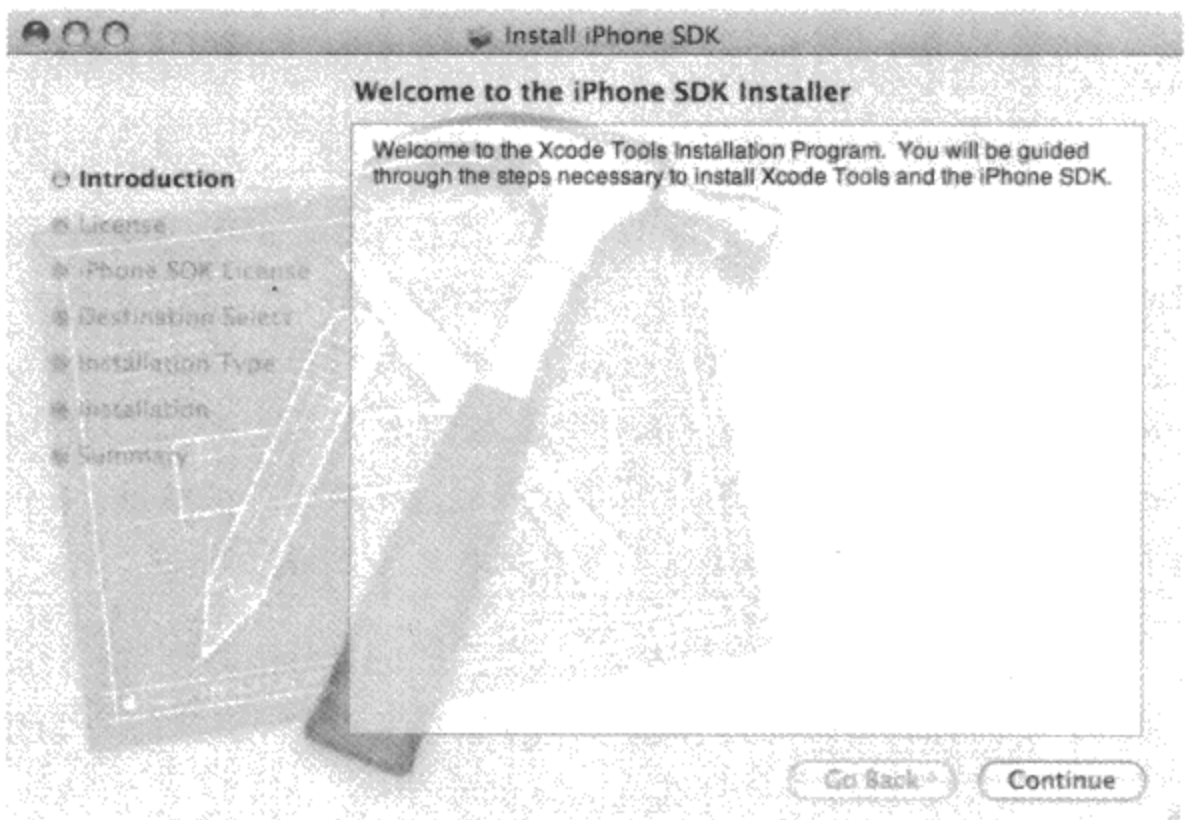


图 9-2　iOS SDK 安装欢迎界面

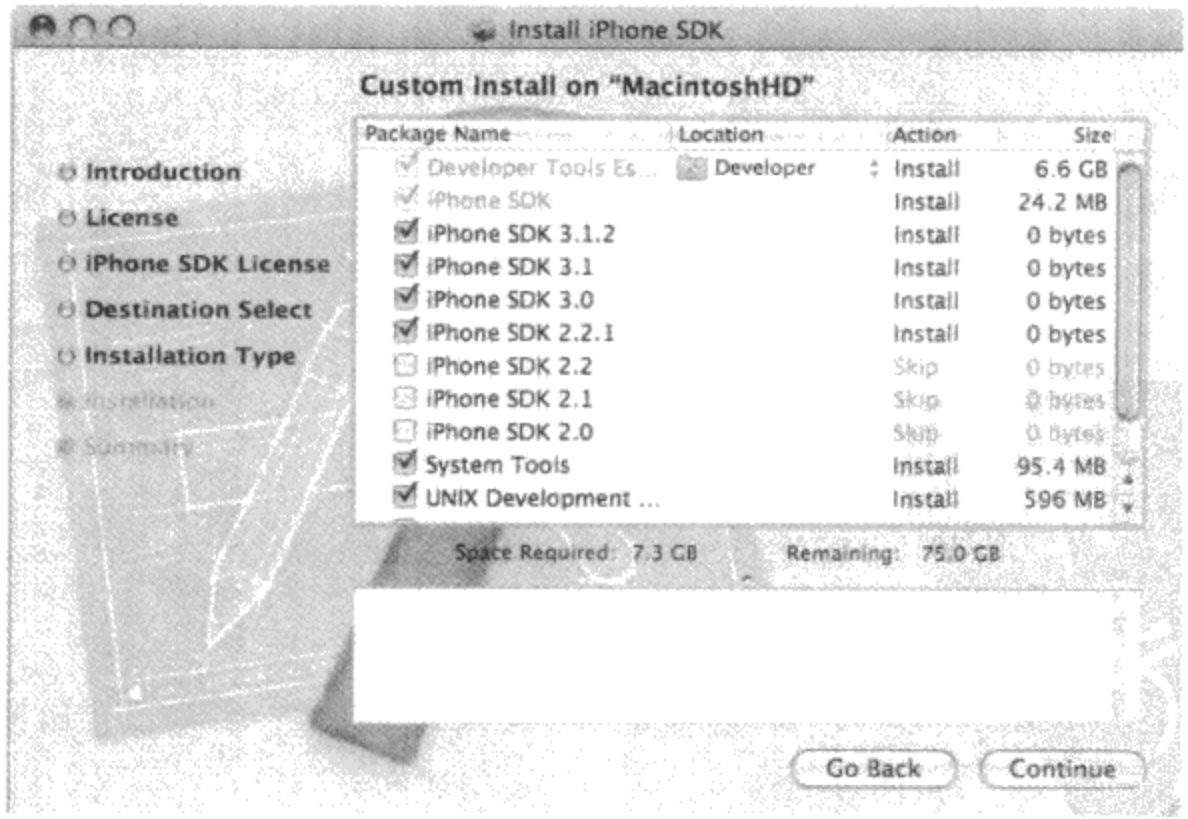


图 9-3　iOS SDK 安装类型

这里，iOS SDK 的安装文件会提示安装文件中包含以前版本的 SDK。安装时除了

总是包含最新版本的 SDK 外，可以选择其他更老的版本。需要这样做吗？对于我们在本书后面章节将要介绍的 SQLite 开发实例而言，安装默认版本就可以了。但是如果以后可能在更老版本的 iPhone 和 iPad 上进行开发，在安装时需要包含更老版本的 SDK。尽管这种情况的概率很小，人们为公司或是企业开发程序都要保证能够应用于最新的 iPad 上，同时也需要能够运行在早期的 iPhone 机器上以满足客户对硬件的要求。

选择安装类型后一直单击“Continue”按钮，直到安装结束，将会看到一个安装成功的界面，如图 9-4 所示。

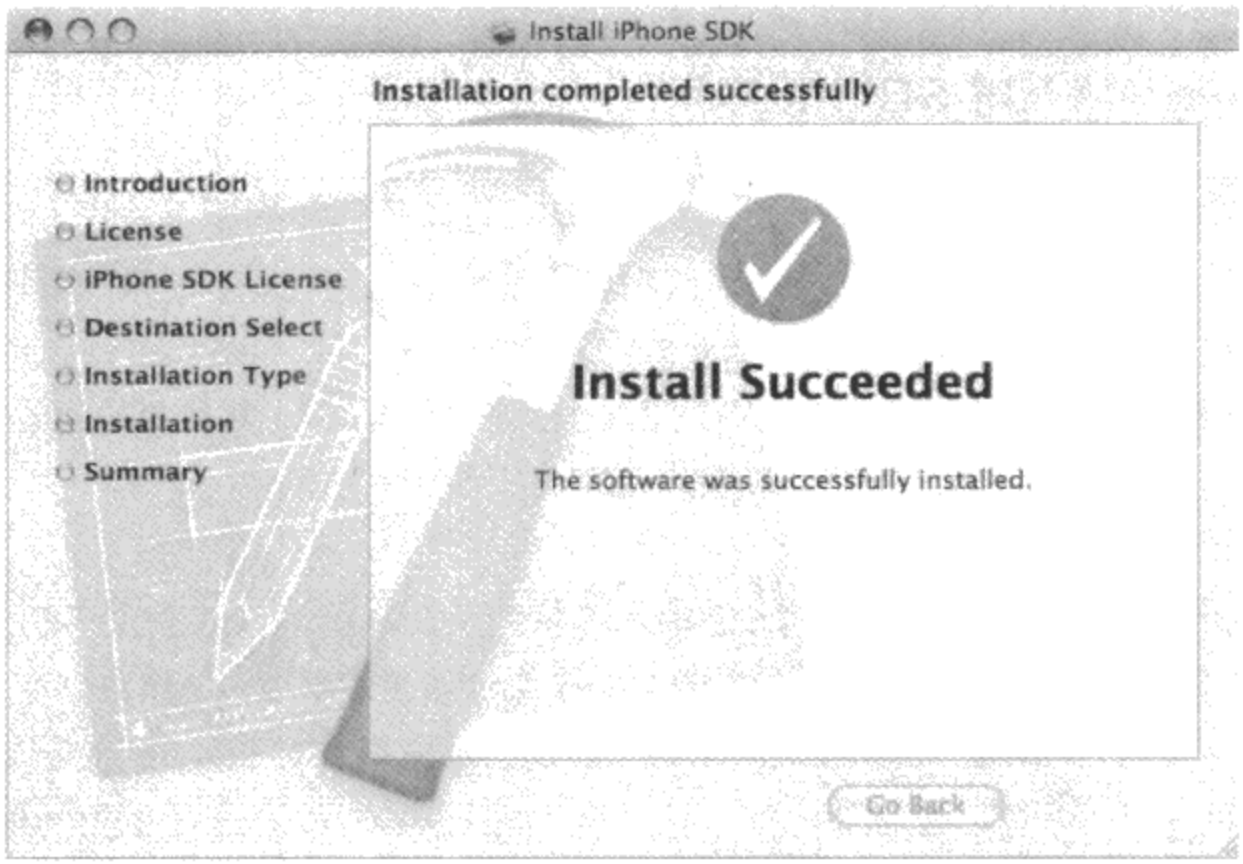


图 9-4　iOS SDK 安装成功的界面

安装成功后，就可以在 iOS 上进行基于 SQLite 的开发了。

其他开发环境

对那些有经验的愿意尝试新开发工具的开发者而言，可以尝试除了 Xcode 外的其他针对 iOS 开发的工具，其中也有很多是基于 SQLite 的 iOS 开发工具。如果已经对 iPhone 开发了如指掌，并且已经习惯了使用某种开发工具，可以随意选择自己熟悉的工具进行开发。下面介绍几种常用的 iPhone 开发环境。

PhoneGap：一个跨平台的开源开发环境，其网址为：www.phonegap.com。

Appcelerator Titanium：另一个开源的跨平台的移动开发环境，其网址为：www.appcelerator.com。

259 以上两个开发环境都能很好地支持 iOS 和其他平台上的 SQLite 开发。苹果公司刚刚放宽了那些之前被人们认为有些反复无常、过于严格的规定，这些规则惩罚那些使用不同语言、不同平台进行开发以及跨平台编译成 Objective-C 和 iOS 的编码。所以，现在有了更多的选择余地，使用以上的工具开发更好的基于 SQLite 的 iOS 应用程序。

建立 iSeinfeld iOS SQLite 应用

搭建好开发环境后，就可以开发基于 SQLite 的 iOS 应用程序了。作为一个出色的程序员，为了更合理地设计应用程序和对何时完成程序心中有数，需要知道用户对应用程序的需求。以下是一些简短的 iSeinfeld 系统要求：

- 为 iPhone 和 iPad 建立一个 iOS 应用程序，按照 Seinfeld 食品受欢迎的程度显示。
- 对于热门食品，显示该食品在所有 Seinfeld 食品中被提及的频率。
- 食物第一次出现时给出提示。

这些要求看起来很简单。我们开发的代码将涉及在 iOS 系统上使用 SQLite 的一些重要方面，包含打开数据库，或者在数据库不存在时创建数据库；操作数据库中的数据；以及所有重要的 UI 设计，这些 UI 设计的目的是引起使用者对 SQLite 数据的关注。

本章包含以下步骤。我们将尽量少地涉及 Objective-C 和 Cocoa 框架的细节内容，重点描述涉及的所有有关 SQLite 的重要方面。

1. 在 Xcode 中创建工程。
2. 添加 SQLite 框架到工程。
3. 准备包含在 iOS 应用程序中的食品数据库。
4. 定义类，将 SQLite 中的食品数据映射到我们的代码中，并且在屏幕上将它们呈现给用户。
5. 访问和查询 SQLite 数据库。
6. 发布应用程序前为它润色。

下面我们将一一介绍这些步骤。

第一步：创建一个新的 Xcode 工程

运行 Xcode。如果这是第一次启动 Xcode（或者没有改变过默认设置），那么将会在 Xcode 的开发环境中看到一个 Xcode 欢迎界面。忽略这个界面——事实上，如果不希望它再次出现的话，取消勾选“在启动时显示”选项。

Xcode 运行后，选择“File”→“New Project”。对于 iSeinfeld 应用程序，需要使用基于向导的应用程序作为工程模板，因为我们将会用到模板和详细的显示来代表各种各样的 SQLite 食品数据。图 9-5 显示了 Xcode 新建工程的选项。 260

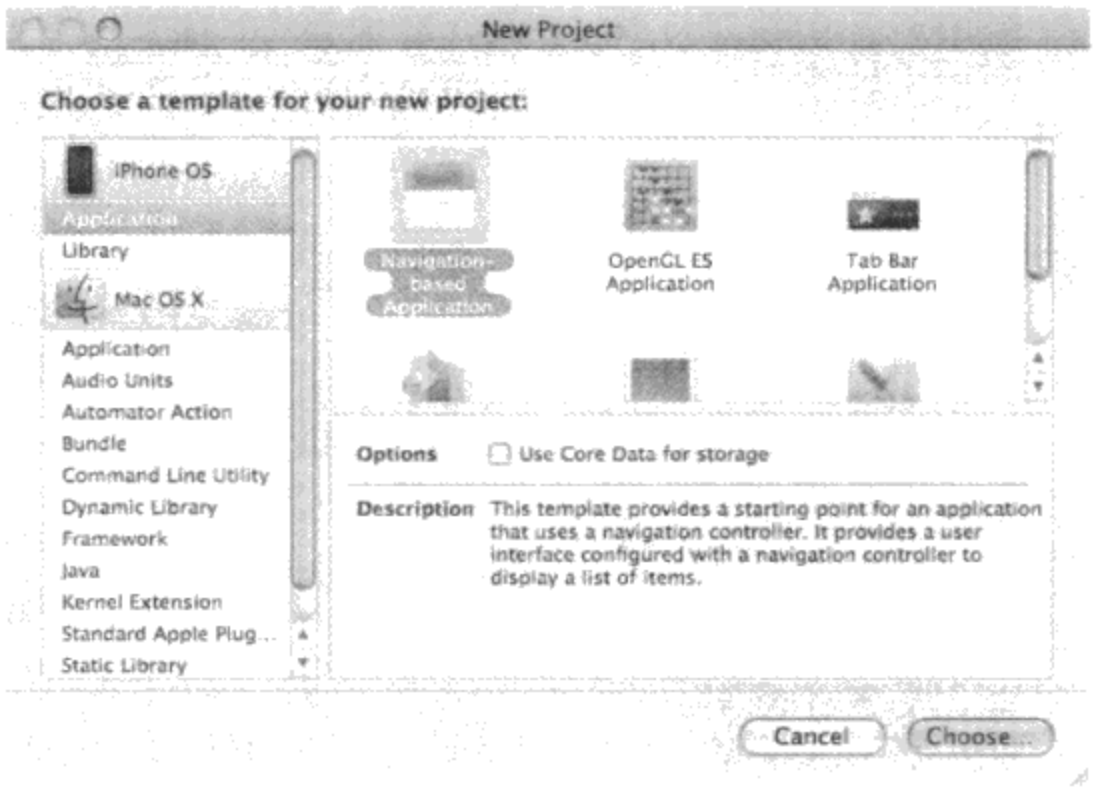


图 9-5　创建一个新的基于定位应用的 SQLite 工程

为工程选择一个有意义的名字。之前一直称我们的工程为 iSeinfeld，但是我们可以采用任何容易记住的或者有意义的名字。默认情况下，这个名称将会是应用程序最终在 iOS 设备上显示的名称（尽管这个名称可以在以后使用时改变）。创建好的工程应该类似图 9-6 中所示。

第二步：将 SQLite 框架添加到工程

在创建了一个空的 iSeinfeld 工程后，需要将 SQLite 库添加到工程框架中，添加这些库后，就能够更加方便地使用 SQLite 的 API 进行开发了。

261

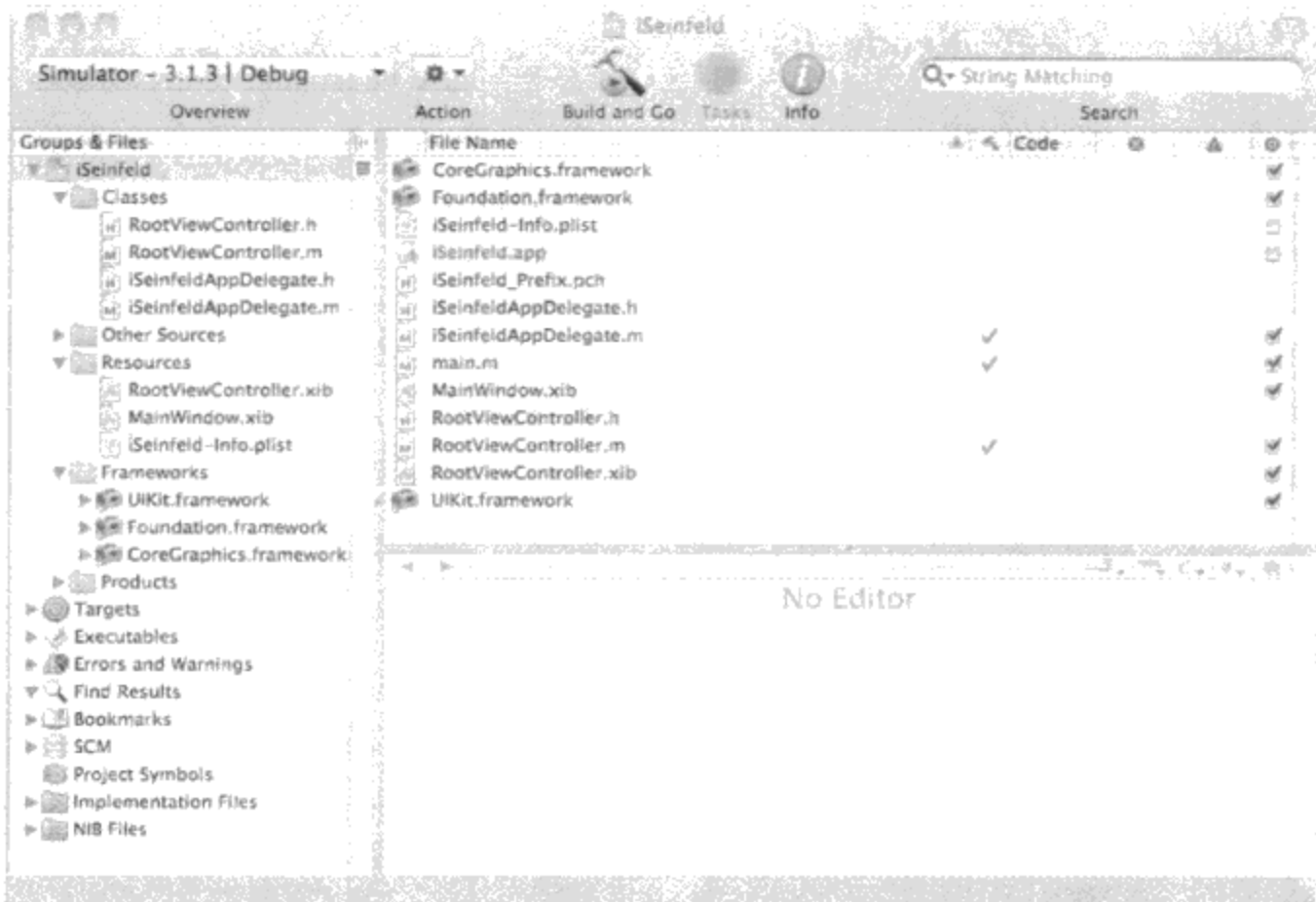


图 9-6　Xcode 中新创建的 iSeinfeld 工程

从 Xcode 工程的左边框里打开“Frameworks”文件夹。按住 Ctrl 键的同时单击“Add”→“Existing Frameworks”，在弹出的对话框中，定位文件夹到/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOSx.y.z.sdk/usr/lib，这里 x.y.z 是 Xcode 和 iOS 所对应的 iPhone 硬件版本号。根据您在安装 Xcode 和 iOS SDK 时的选择，这里可能会看到很多版本的目录。我们推荐选择最新版本（版本号最高的版本）。

262

注意：如果使用的是最新的 iOS 4 版本，那么这些目录显示的是 iOS 而非 iPhone。别担心，它们的底层库是一致的。

在上面的目录中选择 libsqlite3.dylib 项，将会看到名为 libsqlite3.0.dylib 或类似的文件。libsqlite3.dylib 项是一个软链接，链接到最新版本的 dylib 文件，这里就是 libsqlite3.0.dylib。推荐选择链接以避免不必要的版本选择。

选择这个文件后，会弹出一个选项对话框，询问引用类型和文件编码。这里只有一个选项可供选择，所以可以直接单击“Add”选择默认选项，然后返回到 Xcode 工程。这时在左边的“Frameworks”文件夹里应该能看到 libsqlite3.dylib 项，如图 9-7 所示。

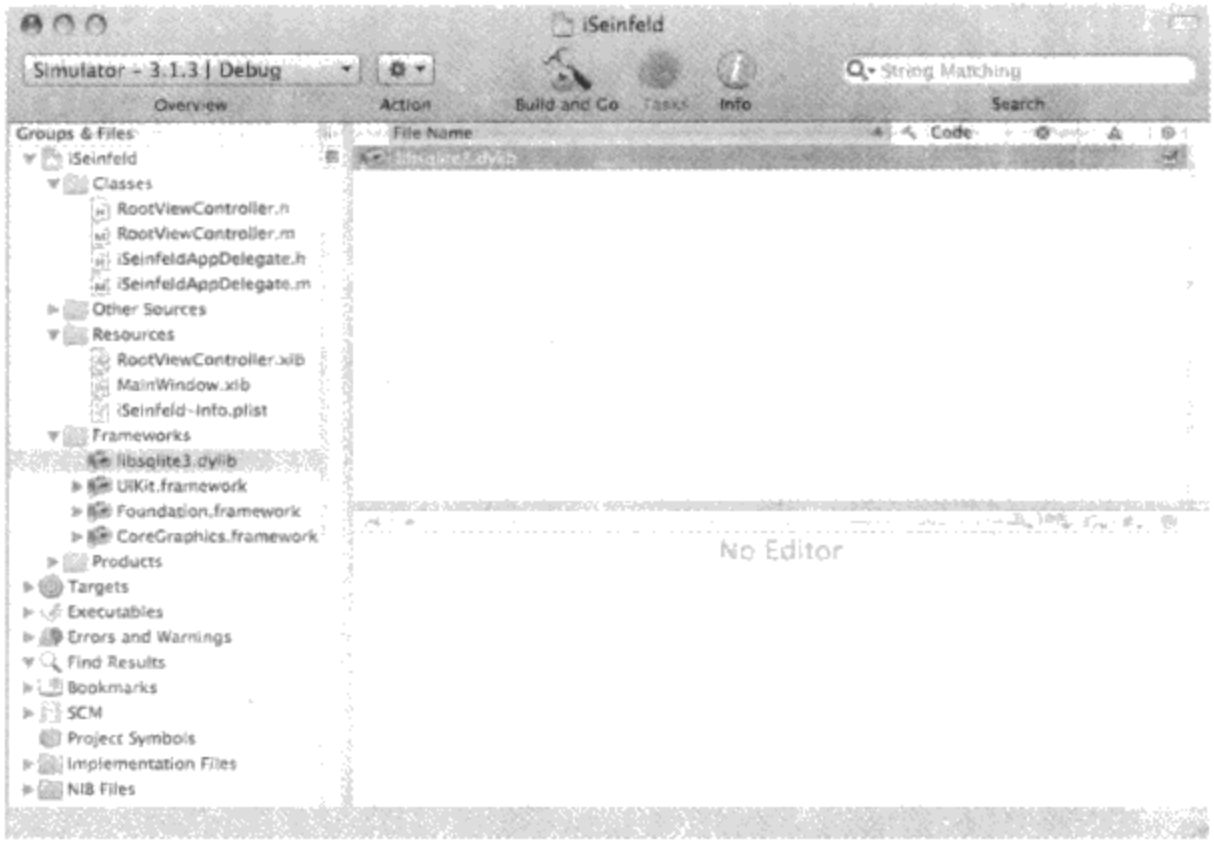


图 9-7　SQLite 的 libsqlite3.dylib 库被加到 Xcode 的 iSeinfeld 工程中

第三步：准备 Foods 数据库

我们有两种在工程中可以使用的数据库方案：第一种方案是，按部就班地创建一个新数据库、创建表、插入数据；第二种方案是，可以在 iSeinfeld 应用程序中使用一个已经存在的数据库。

可以通过 SQLite API: sqlite3_open_v2()、sqlite3_prepare()等一步步地手工创建数据库，不停地重复执行 sqlite3_exec()函数。因为 C 和 Objective-C 非常接近，创建数据库的步骤在第 6 章已经介绍了，这里就不再重复。更重要的是，当建立和发布下一个更好的 iOS 应用程序时，很可能是将它与一个已有数据库打包在一起（虽然可能会提供一些在已有数据上添加新数据的功能）。我们采用第二种方案，但是也会介绍如何创建新数据库，以及一些会在 SQLite 中用到的典型的操作数据的 SQL 命令。下面我们将已有的 foods.db 数据库添加到工程中，这样可以更加便捷地使用数据。

如果使用 Xcode 中的默认路径，那么 iSeinfeld SQLite 工程将在目录/Users/user_name/Documents/iSeinfeld 文件夹下（这里 user_name 是当前用户名）。复制 food.db 文件到这个目录下以保证工程数据、代码和资源文件在一起。在 Xcode 工程左边的面板中，打开资源文件夹。按住 Ctrl 键的同时单击“Add”→“Existing files”。在添加文件对话框中，

选择“/Users/user_name/Documents/iSeinfeld”目录，然后选择 foods.db 文件，如图 9-8 所示。

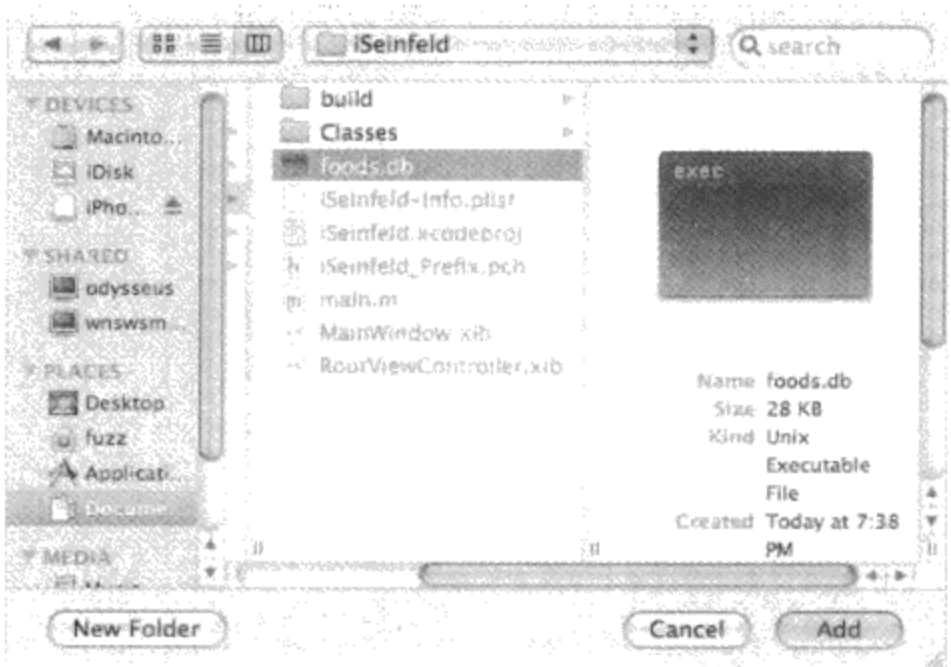


图 9-8　添加已有 SQLite 数据库到 Xcode 工程

强调一下，Xcode 在添加文件到工程时会提示多个选项。这里我们采用默认设置，直接单击“Add”，将 foods.db 文件添加到工程中。此时，资源文件夹的列表里会出现 SQLite 数据库 foods.db，如图 9-9 所示。

264

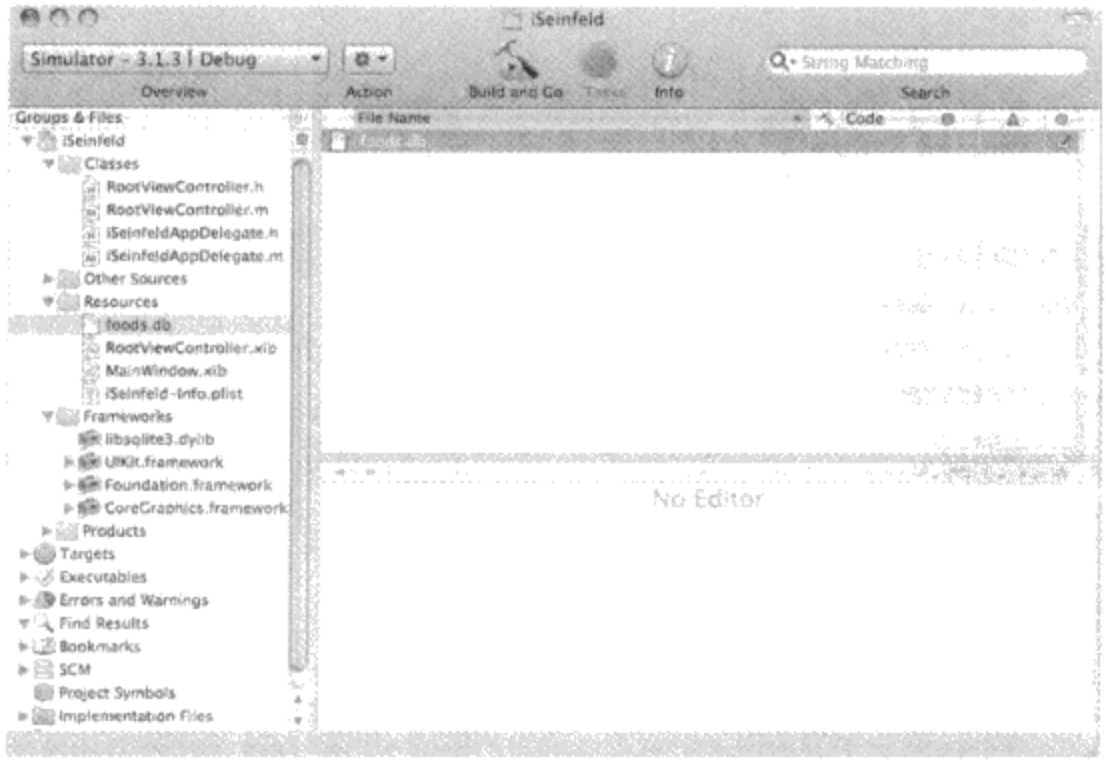


图 9-9　iSeinfeld 工程添加了 foods SQLite 数据库资源

至此，iSeinfeld iOS 应用程序已经在工程中添加了 SQLite 库和一个数据库。现在可以开始编码工作了。

第四步：为食品数据创建类

除了新建工程中默认创建的类外，还需要两个类来表示应用程序中 SQLite 数据库的数据。首先，需要一个 Food 类来表示食品的类型和应用程序中的食品数据。其次，需要一个 controller 类来管理用户界面与数据模型之间的交互。

iOS“模型-视图-控制（MVC）”开发模式

“模型-视图-控制”开发模式将一个应用程序按照功能进行分隔。近些年，MVC 模式变得越来越成熟和流行，这种模式将使 iOS 方法更加专注于开发，Xcode 工具则在 MVC 模型中担任协调工作。

我们举的小例子不足以显示 MVC 工具包中的所有功能以及它和 Xcode 结合工作的模式，但是我们可以保证 Xcode 对于 MVC 开发提供了足够的支持和方法。

Food 类

在 Xcode 的 iSeinfeld 工程中，单击 Classes 文件夹，选择“Add”→“NewFile”，为新类命名为 Food。我们希望 Food 类从 Objective-C 基类中继承，并且使用 NSObject 作为子类。设定三个数据成员：Name、Popularity 和 First Episode。另外定义一个 init 函数（考虑到为其他面向对象语言提供一个类似的构造函数）创建 Food 类实例。代码 9-1 列出了 Food.h 头文件的内容。

代码 9-1　Food.h

```
// Food.h
// iSeinfield
//
// Grant Allen on 于 2010 年 9 月 9 日创建。
// 版权、公司名称、2010 版权所有。
//

#import <Foundation/Foundation.h>

@interface Food : NSObject {
```

```
  NSString *first_episode;
  NSString *popularity;
  NSString *name;
}

@property (nonatomic, retain) NSString *first_episode;
@property (nonatomic, retain) NSString *popularity;
@property (nonatomic, retain) NSString *name;

-(id)initWithName:(NSString *)n first_episode:(NSString *)f popularity:(NSString *)t;

@end
```

从上面的头文件中看不到 SQLite API 调用迹象，但是也许您已经发现了 Food 类已经准备好了。SQLite 查询返回的结果可以用做 initWithName 函数的种子数据来创建 Food 类对象的实例。

266

准备好头文件后，接下来需要实现头文件中的函数。打开 Food.m 文件，如代码 9-2 所示，initWithName 函数实现了将参数 n、f 和 t 赋值给实例变量的功能。

代码 9-2　Food.m

```
// Food.m
// iSeinfield
//
// Created by Grant Allen on 9/9/10.
// Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import "Food.h"

@implementation Food
@synthesize first_episode, popularity, name;

-(id)initWithName:(NSString *)n first_episode:(NSString *)f popularity:(NSString *)t {
  self.name = n;
  self.first_episode = f;
  self.popularity = t;

  return self;
}
```

```
@end
```

即使以前从未使用过 Objective-C 编程，也能理解上面的代码。对于那些 Objective-C 新手而言，需要注意@synthesize 开始的一行，该行中定义的都是这个类的变量。这是 Objective-C 的一个技巧，即自动为变量创建 set 和 get 函数（setters and getters），省去了手动书写 6 个空函数的麻烦。

FoodViewController 类

iSeinfeld 应用程序要求显示 SQLite 数据库中所包含食物的详细信息。在 Food 类中，您可能已经猜到了提供详细信息的是 popularity 和 first_episode 函数。我们还需要一个类管理这些具体内容的视图（按照 MVC 的说法），并且控制 MVC 模型。FoodViewController 类中同样需要一个 XIB 视图文件作为画布展现细节。在 Resources 下，选择“Add”→“New File”，然后从用户接口组中选择视图 XIB，并将这个 XIB 文件命名为 FoodViewController.xib，以提醒这个 UI 是与 FoodViewController UIViewController 类相关联的。

FoodViewController 类需要定义两个与 XIB 交互数据的接口。代码 9-3 显示了 FoodViewController.h 文件，以及它定义的函数。

代码 9-3　FoodViewController.h

267

```
// FoodsViewController.h
// iSeinfield
//
// Created by Grant Allen on 9/9/10.
// Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import <UIKit/UIKit.h>


@interface FoodViewController : UIViewController {
  IBOutlet UITextView *first_episode;
  IBOutlet UITextView *popularity;
}

@property (nonatomic, retain) IBOutlet UITextView *first_episode;
@property (nonatomic, retain) IBOutlet UITextView *popularity;
```

```
@end
```

因为需要展示文本数据和数字，我们将 UITextView 作为布局修改器。FoodView Controller.m 实现了 FoodViewController 类，包含了一些默认函数和许多注释掉的内容。这里，我们就不浪费篇幅一一介绍它们了。只要保证在 FoodViewController.m 中引入了头文件，并且保证和头文件中定义的变量一致就可以，像下面这样：

```
#import "FoodViewController.h"

@implementation FoodViewController

@synthesize foodID, typeID;

// 更多的模板方法，等等。
// …
```

FoodViewController.xib 的布局完全取决于您的设计。作为介绍，我们只是简单地创建了一个窗口，包含两个文本框和两个标签，并利用接口建造器将它们与刚刚定义的接口捆绑在一起。图 9-10 显示了 XIB 定义的视图。

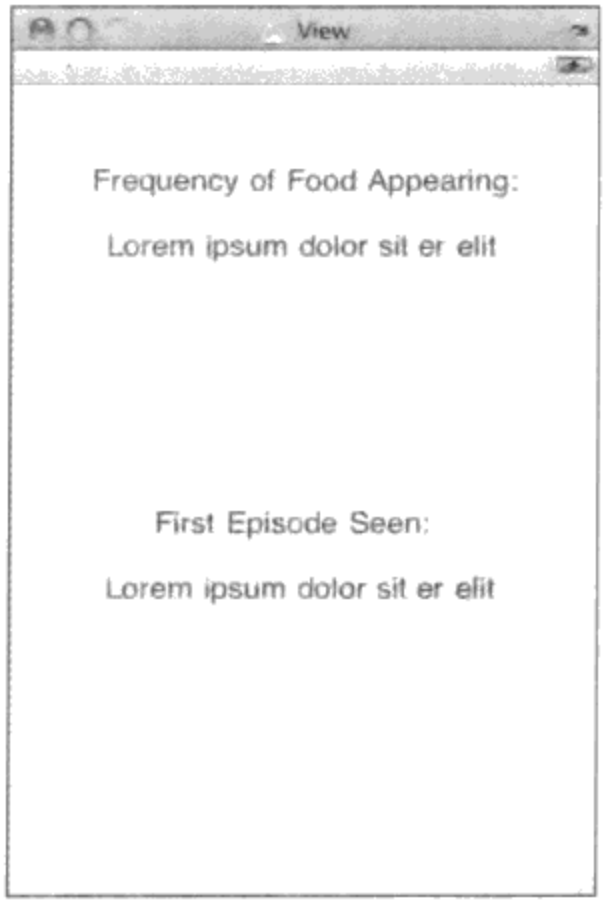


图 9-10　Xcode 接口建造器中现实的 Fruit 细节的截屏

现在 iSeinfeld iOS 应用程序可以显示 SQLite 数据库中的数据了，但是目前还是不能真正使用 SQLite API 访问数据库中的数据。下面我们访问数据库。

第五步：访问和查询 SQLite 数据库

启动 iSeinfeld 应用程序后，想让程序打开 food.db 数据库。这时，可以使用 SQLite 的 C API 对数据库进行访问和控制。为了在启动时能够找到并打开数据库，需要扩展工程自动生成的 iSeinfeldAppDelegate 类。代码 9-4 显示了对 iSeinfeldAppDelegate 头文件所做的修改。

代码 9-4　iSeinfeldAppDelegate 头文件

```
// iSeinfeldAppDelegate.h
// iSeinfeld
//
// Created by Grant Allen on 9/9/10.
// Copyright __MyCompanyName__ 2010. All rights reserved.
//

@interface iSeinfeldAppDelegate : NSObject <UIApplicationDelegate> {

  UIWindow *window;
  UINavigationController *navigationController;

    // 数据库的详细内容。
    NSString *databaseName;
    NSString *databasePath;

    // 我们的 iseinfeld 食物数组，对食物进行排序
    NSMutableArray *foods;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UINavigationController *navigationController;
@property (nonatomic, retain) NSMutableArray *foods;

@end
```

上面高亮显示的代码定义了保存和命名 SQLite 数据库的变量。另外，还定义了保

存 foods 集合的 NSMutableArray。这些接口定义一目了然，但是具体实现起来就没那么简单了。

我们将在 iSeinfeldAppDelegate.m 文件中实现类中的方法。打开 iSeinfeldAppDelegate.m 文件将会看到很多自动生成的函数，比如 applicationWillTerminate 等。我们需要在程序启动时就调用应用程序中的函数，通过修改 applicationDidFinishLaunching 函数可以实现。

```
// 上面有很多代码，此处省略
- (void)applicationDidFinishLaunching:(UIApplication *)application {

  // 将 foods 数据库作为 databaseName
  databaseName = @"foods.db";

  // 获取文档目录的路径并添加到数据库名前面
  NSArray *documentPaths = NSSearchPathForDirectoriesInDomains(
                              NSDocumentDirectory, NSUserDomainMask, YES);
  NSString *documentsDir = [documentPaths objectAtIndex:0];
  databasePath = [documentsDir stringByAppendingPathComponent:databaseName];

  // 检查数据库是否存在，如果需要，则创建新的
  [self checkAndCreateDatabase];

  // 查询数据库的所有食物，并将数据填充到数组中
  [self readFoodsFromDatabase];

// 后面还有很多代码，此处省略
```


我们使用很多工程内置的功能寻找 SQLite 文件 food.db。NSSearchPathForDirectories InDomains、NSDocumentDirectory 和 NSUserDomainMask 都是 iOS SDK 提供的寻找运行环境中路径的常量。本例中，这些代码都是定位当前应用程序位置的有效方法，不管应用程序在什么位置。代码最后的 databasePath 表示了 foods.db 文件所在路径。

我们的思路就是将代码分成两部分，即查找数据库（或者在工程中重新创建数据库）和访问数据库。查找数据库部分的代码并不是 SQLite 特有的，它覆盖了很多简单的文件操作。iSeinfeldAppDelegate.m 文件中包含了这部分代码。

打开数据库，然后利用 SQLite API 访问数据是需要实现的部分。这里 readFoods-FromDatabase 函数实现了这部分功能，下面具体介绍。

```
-(void) readFoodsFromDatabase {
  // 建立数据库对象
  sqlite3 *database;

  // 初始化数组 foods
  foods = [[NSMutableArray alloc] init];

  // 打开数据库
  if(sqlite3_open_v2([databasePath UTF8String], &database) == SQLITE_OK) {
    // 建立 SQL 语句，为了快速访问并编译该语句
    const char *sqlStatement =
      "select
       count(foods_episodes.episode_id) popularity,
       min(foods_episodes.episode_id) first_seen,
       foods.name
       from foods inner join foods_episodes on foods.id = foods_episodes.food_id
       group by foods.name
       order by popularity desc, name
       limit 50 ";

     sqlite3_stmt *compiledStatement;
```



```
    if(sqlite3_prepare_v2(database, sqlStatement, -1, &compiledStatement, NULL)
      == SQLITE_OK) {
      // 循环遍历结果，并将它们添加到数组 foods 中
      while(sqlite3_step(compiledStatement) == SQLITE_ROW) {
        // 从结果行读取数据
        NSString *aFreq = [NSString stringWithUTF8String:
          (char *)sqlite3_column_text(compiledStatement, 0)];
        NSString *aPop = [NSString stringWithUTF8String:
          (char *)sqlite3_column_text(compiledStatement, 1)];
        NSString *aName = [NSString stringWithUTF8String:
          (char *)sqlite3_column_text(compiledStatement, 2)];

        // 用来自数据库的数据创建新的 food 对象
        Food *food = [[Food alloc] initWithName:aName
                        first_episode:aFreq popularity:aPop];

        // 将 food 对象添加到 foods 数组中
        [foods addObject:food];

        [food release];
```

```
        }
    }
    // 结束语句
    sqlite3_finalize(compiledStatement);

  }
  sqlite3_close(database);
}
```

这段代码中包含很多熟悉的 API 调用。开始定义了一个名为 database 的 sqlite3 对象：

```
sqlite3 *database;
```

在为 foods 定义了一个数组后，试图调用 sqlite3_open_v2()这个 API 函数打开数据库。

```
if(sqlite3_open_v2([databasePath UTF8String], &database) == SQLITE_OK)
```

不管您之前习惯使用哪种语言，对 if 结构应该不会陌生。if 条件是判断 sqlite3_open_v2()函数的返回值是否等于 SQLITE_OK。在第 5～7 章中已经介绍过类似的例子了。值得注意的是，处理 databaseName 时的不同之处，之前的例子中使用 iSeinfeldAppDelegate 变量记录数据库的绝对路径和文件。因为 Objecive-C 使用 NSString 作为字符串类的父类，但是 SQLite 的 C API 使用标准的 C 模式的字符串和字符，所以需要将 NSString 类型的 databaseName 转换成 C 字符串，只有这样，SQLite API 才能得到需要打开的数据库的名字。内置的 UTF8 字符串选项为我们完成了这种转换。

当数据库打开后，创建了一个名为 compiledStatement 的 sqlite3_stmt 对象，用该对象保存查询文本。

```
sqlite3_stmt *compiledStatement;
```

这些代码和 C 语言的语法极为相似：Objective-C 和 C 之间的区别不会阻碍您使用它编程。我们使用 sqlite3_prepare_v2 保存查询语句，使用 sqlite3_step 保存中间的查询结果。

我们使用下面的语句检索数据库中的数据：

```
NSString *aName = [NSString stringWithUTF8String:
        (char *)sqlite3_column_text(compiledStatement, 2)];
```

在这里您会发现一个非常熟悉的函数：调用 sqlite3_column_text() API 来得到“name”

列的内容。我们可以根据情况使用任何 sqlite3_column_xxx()函数。接下来，我们将 C 中的 char*类型数据转换成 Objective-C 的 NSString 类型。stringWithUTF8String 完成了这个转换，并把结果保存在 aName 变量中。对其他结果行也采用同样的处理步骤。

到这里，每一个 Food 对象都已经添加到了 Foods 数组中，继续这个处理过程，直到处理完所有的结果。最后，使用 sqlite3_finalize()函数销毁对象，利用 sqlite3_close()函数关闭数据库。注意，这个例子中，我们关闭了数据库，因为已经完成了所有要执行的工作。在更复杂或者功能更多的应用程序中，为了以后的查询，可以保持数据库打开，或者选择在收到 applicationWillTerminate 事件时，关闭数据库。

第六步：最后包装和配置 iSeinfeld 应用程序

最后，需要几个配置步骤让我们的应用程序按照要求运行 FoodView，查看食物的详细信息。这里我们就不详细介绍这部分代码了，它们不是 SQLite 特有的代码，您可以参考包含在源代码中的文件。我们已经很清楚地在注释中标注了代码的变化，您可以通过对比 Xcode 中的 vanilla 文件来查看这些改动。

运行 iSeinfeld

悬念已经揭晓了！我们已经准备好代码，可以访问 SQLite 数据库了，并且搭建了清晰的用户接口显示 Seinfeld。现在可以在 iOS 模拟器中运行新应用程序了。选择“Build”→“Build and Run”，启动设备模拟器，例如 iPhone 的设备模拟器如图 9-11 所示。

继续下一步，单击新应用程序运行它，会看到发生了什么，主屏显示的内容如图 9-12 所示。

提示：如果想通过不同的任务查看之前的 SQLite API 调用，需要在 iSeinfeldAppDelegate.m 文件的 readFoodsFromDatabase 函数中设置断点。

选择任意一个 foods 列表中的食物，进入 FoodView 视图，该视图由 FoodViewController 控制，用来显示所选食物的具体信息。图 9-13 中显示了热狗的详细信息——您知道吗？热狗是 Seinfeld 历史中最流行的一种食物。

273
274



图 9-11　iOS 设备模拟器中基于 SQLite 的 iSeinfeld 应用程序图标



图 9-12　iSeinfeld 首页显示了 foods.db SQLite 数据库中的 foods 列表

275



图 9-13　iSeinfeld FoodView 截屏，显示了数据库中热狗的详细信息

我们的界面设计并不是很出色。但是有了这个应用程序，就可以在生活中利用 iPad 和 iPhone 这些设备访问您的 SQLite 数据库了。至此，我们对 iSeinfeld 的介绍告一段落了。 276

关键数据的研究

本书介绍了 SQLite 以及如何使用它解决各种问题。但是有时候需要抽象出 SQLite C API 解决更抽象的问题。我们原本打算介绍关键数据——iOS 3 以后版本中都包含的数据管理抽象层，但是我们没有这样做，只是展示了 iOS 中利用 SQLite 工作的核心部分。

如果您想了解更多在 iOS 中 SQLite 数据库的访问和管理的数据抽象层的知识，可以借助 http://developer.apple.com/library/ios 提供的关键数据文档。

iOS 中处理大型 SQLite 数据库

敏锐的人已经注意到了统计访问 iSeinfeld 应用程序频率的查询包含下面一句代码：

```
limit 50
```

为什么要限制查询结果的数量呢？我们这样做是为了简化应用程序，避免在 iOS 中处理大型的 SQLite 数据库。试图在 applicationDidFinishLaunching 事件中处理大型数据库，可能会引起 Watchdog Timer 错误。Watchdog Time 的作用是保证所有运行在 iPad、iPhone 或者 iPod 上的应用程序正常运行，包括阻止运行异常的占用过多处理器而使设备挂起的应用程序。

苹果公司在 Apple iOS 开发者文档中提到 Watchdog Timer 是一个系统监视器，监测应用程序需要花费多长时间完成启动。如果花费了太长的时间，那么 Watchdog Timer 就会终止该应用程序，并抛出异常错误 0x8badf00d。相关的信息会写在错误报告中。

在 Xcode 中运行代码时，Watchdog Timer 是禁用的，以补偿 OS X 系统中的各种开支，比如调试器等。这就意味着，在 Xcode 模拟器中，您可能注意不到 Watchdog Timer。最好在真实的设备中进行测试，检验代码长期运行时可能出现的问题。

在 iOS 中设计 SQLite 应用程序的常用管理方法可以归纳为以下几点： 277

- 不要在启动事件中（例如 applicationDidFinishLaunching）运行耗费时间的查询，也不要加载大型数据库。

- 当查询结果很大，或者查询要耗费很长时间时，考虑把查询分解成可运行的几个部分分别查询。
- 运用更加高级的 iOS API 功能管理需要长时间运行的任务。

最后，如果 Watchdog Timer 决定中断应用程序，那么将会返回如下错误：

```
The application iSeinfeld quit unexpectedly

Mac OS X and other applications are not affected
```

在真实的 iPad 或者 iPhone 中，不会出现这个错误信息，而是看到应用程序试图加载首页但尝试失败，这时可以返回设备主页。了解了 Watchdog Timer 的功能后，就尽量不要去触发它。

总结

现在您已经尝试了在 iOS 中创建第一个基于 SQLite 的应用程序。您的家人和朋友会对您新学到的知识感到惊叹，他们也许会对如何简便地使用 SQLite 创建自己的 iPad 或者 iPhone 应用程序感兴趣。

如果您决定制作一个 iSeinfeld 应用程序，请一定要让我们知道。我们愿意从 iTunes 上购买它，并添加到 iSeinfeld 家族中。

第 10 章

Android 开发中的 SQLite

Android 系统上市时间虽短，但是已经席卷了整个移动设备市场。如果您认为 Android 系统还不错，并购买过手机、平板电脑或者其他运行 Android 系统的设备，那么您已经是 SQLite 用户了！SQLite 是 Android 平台内置的一项数据持久技术，同时 Dalvik 虚拟机也支持 SQLite。与第 9 章介绍的 iOS 系统不同，Dalvik 是基于 Java 的，这就意味着开发者如果要在应用程序中使用 SQLite，需要一个特制的 API 封装器在外层封装 SQLite C API。

本章中，我们将把 Android SQLite 库、类和方法放在一起进行介绍。与第 9 章中介绍的 iOS 系统一样，我们将搭建一个真实的可以工作的基于 SQLite 的 Android 应用程序，以此进行讲解，而不是简单地重复文档。我们将沿用 Seinfeld 食物应用程序，介绍如何进行代码的编写和界面设计。

另外，还会介绍一些在 Android 设备上长期维护应用程序时必须考虑的因素。

SQLite Android 开发的先决条件

在 Android 系统上进行 SQLite 开发时，必须具备一些工具。这些工具不仅可以在 Android 平台上进行 SQLite 相关的开发，还可以开发其他 Android 应用程序。显然，SQLite 开发是我们感兴趣的内容，所以会把重点放在 SQLite 上。

Android 系统开发的准备工作可以分为四步：

1. 检查并安装包括 JDK 在内的开发所需工具。
2. 下载并安装 Android SDK starter 包。
3. 下载并安装集成开发环境所支持的 Android 程序开发工具（ADT）。

4．在环境中添加需要的 Android 平台和其他组件。

在介绍 SQLite 之前，先简短介绍以上四步。

280 检查先决条件和 JDK

在开发任何 Android 应用程序前，都要检查用于开发的机器是否满足系统的几个要求。首先，必须能够运行 Linux、OS X 或者 Windows XP 及其更新版本的 Windows。Android 开发团队列出了每个版本对应的要求，您可以在 http://developer.android.com/sdk/requirements.html 上查到。另外，除了 IDE 所需的硬盘空间外，Android 组件需要 500MB 的硬盘空间。

JDK 是最重要的先决条件。Android 开发要求 JDK 5 或者 JDK 6。注意，Java 运行环境（JRE）是不行的，必须是 JDK。您可以从 Oracle 网站上下载最新的 JDK，或者选择包含 JDK 的 IDE。

所有的 SQLite Android 应用程序都将在 IDE 中进行开发。有很多 IDE 可以选择，但是初学者最常用、最好用的 IDE 是 Eclipse。Eclipse 中还可以使用 Google 提供的 Android 开发工具，我们稍后会介绍这些工具。Android 团队推荐使用 Eclipse 3.4 或更高版本，并且选择“classic”或者“java-focused”版本。

下载和安装 Android SDK Starter 包

Android SDK Starter 包是 Android 开发所需的重要工具。重要的是，它不是一个完整的 SDK。更确切地说，它更像是一个地基，指导用户下载其他必需的组件。

从 Google 上下载 Android SDK Starter 包很容易，登录网站 http://developer.android.com/sdk/index.html，可以找到最新版本的 SDK 进行下载。在撰写本书时，发布的最新版本是 7。Linux、OS X 或者 Windows 需要不同的安装包：

- Linux：android-sdk_r07-linux_x86.tgz。
- Mac OS X：android-sdk_r07-mac_x86.zip。
- Windows：android-sdk_r07-windows.zip。

下载 SDK 后，就可以进行安装了。解压安装包.zip 或者.tgz 文件到计算机中的某个路径。例如，在 Windows 中使用 Eclipse，将压缩的 SDK 解压到目录 c:\eclipse\dropins\android-sdk-windows。

解压 SDK 到某个路径后，需要将这个路径添加到系统的环境变量中。如果不知道如何添加路径到环境变量 PATH 中，请参考第 2 章。

下载和安装 Android 开发工具

如果选择 Eclipse 作为基于 SQLite 的 Android 开发 IDE，那么除了 Android SDK Starter 包外，还需要下载和安装 Android 开发工具（ADT）。ADT 扩展了 Eclipse，使得开发者可以通过 Android 相关入口快速创建新的 Android 工程，另外还提供了用于设计 UI、调试程序、导出签名或未签名的 Android 应用程序（称为 APK）的工具。

安装 ADT 的步骤为：运行 Eclipse，选择“Help”→“Install New Software”（在 Eclipse 3.5 中是这样，在更早的 Eclipse 版本中，菜单可能会有些不同）。单击“Add”按钮添加新的软件库，输入 Android 开发工具所在的 URL，单击“OK”按钮，返回到安装新软件的界面，应该能够看到开发工具及其子组件 Android DDMS 和 Android 开发工具，如图 10-1 所示。

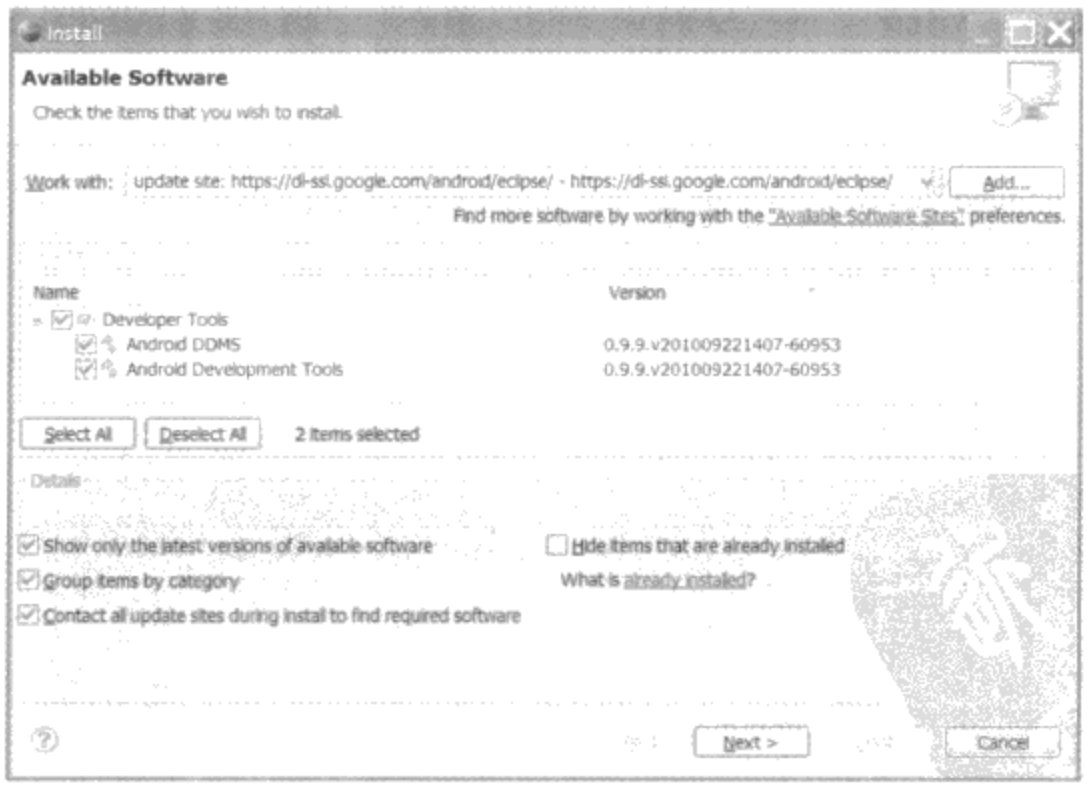


图 10-1　安装 Android 开发工具

选择所有的复选框，然后单击“Next”按钮，出现许可协议界面，同意许可协议之后，单击“Finish”按钮，完成 ADT 的安装。

添加 Android 平台和组件

最后安装 SQLite 开发所需的 Android SDK，Android SDK 和 AVD 管理器负责将其他组件下载到开发环境。Google 将 Android SDK 分成模块，因此，SDK 相关文档，不同平台或版本的组件和手册等都已经分成了不同的安装包进行管理。可以选择以下组件下载安装。

- SDK 工具包：这是 SDK 的核心部分，该安装包已经随 SDK Starter 包安装到开发环境了。
- Android 平台：任何一个 Android 平台的版本对其他已发布的 Android 平台都是适用的。至少需要安装一个平台，以保证基于 SQLite 的 Android 应用程序有一个可运行的平台。
- SDK Add-Ons：SDK Add-Ons 是与 Android SDK 捆绑的插件，它能够提供更多更有用的功能。其中，使用最广泛的就是 Google API Add-On，这个插件可以访问很多 Google 应用，比如地图等。
- Windows USB 驱动程序：USB 驱动程序是为 Windows 平台的程序员准备的。安装该驱动程序后就可以把编写的应用程序放到一个真实的环境中编译和运行。对于 OS X 和 Linux 用户，则不需要这个工具。
- Samples：这个包中包含了很多 Android 开发的示例代码。
- 文档：这个包中包含所有版本的 Android SDK 文档。

在 Eclipse 菜单中选择“Window”→“Android SDK and AVD Manager”添加这些组件。管理器窗口如图 10-2 所示。

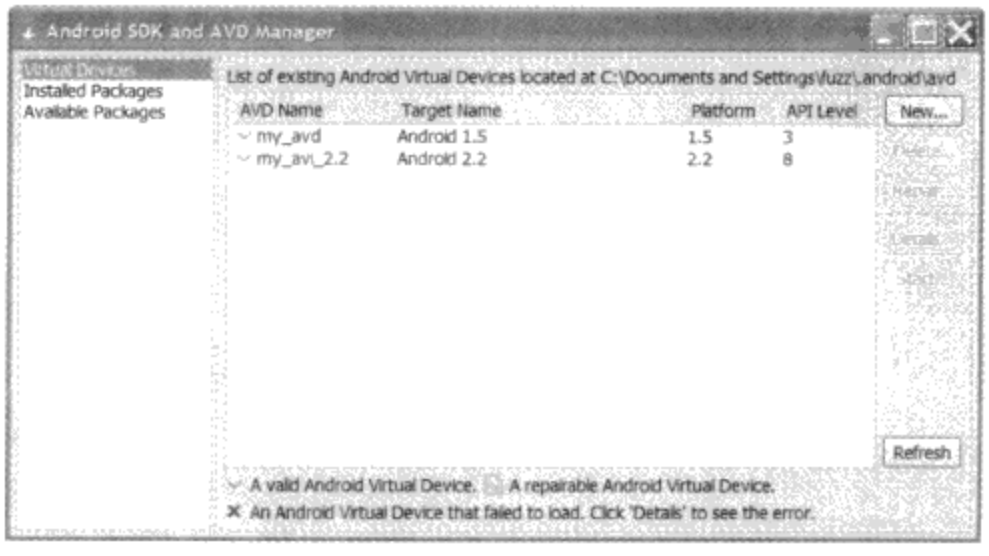


图 10-2　Eclipse 中的 Android SDK 和 AVD 管理器

在图 10-2 的管理器视图中，可能会看到存在一个或多个虚拟设备。下面我们为 SQLite 开发单独创建一个新的 AVD，在这个 AVD 中只添加我们需要的组件，其他多余的组件可以去掉。 283

在 Android SDK 和 AVD 管理器窗口中，单击“New”按钮，弹出新建 AVD 对话框，如图 10-3 所示。

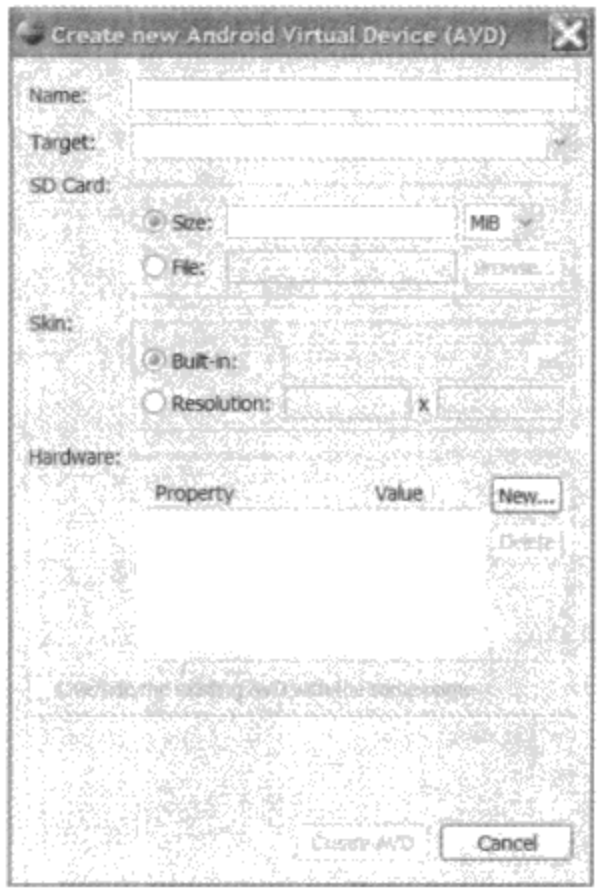


图 10-3　在 SQLite 工程中新建 Android 虚拟机

在图 10-3 的“Name”中为刚新建的虚拟设备命名，这里我们称它为 SQLite_AVD。“Target”域决定了应用程序是为哪个 Android SDK 版本服务的。

Android 版本和 SQLite 版本 284

不同版本的 Android 中包含的 SQLite 库版本也不同。Android 2.1 及其之前的版本中 SQLite 的版本是 3.5.9。从 Android 2.2 之后（称为 Froyo），SQLite 库升级到了 3.6.22。

如果没有什么特殊要求，Android 可以支持使用最广泛的最旧的 SDK 版本。另外，我们推荐使用最新的 Android 2.2。

SD Card 和 Skin 设置不是使用 SQLite 开发时考虑的重点，所以我们使用它的默认设置。虚拟设备需要考虑模拟的其他硬件。这些硬件有不会影响 SQLite 开发的 GPS、

加速表等。但是有些硬件设备的配置需要特别注意。

“缓存大小”控制着设备分配给应用程序和手机用户内存的比例。常用手机和平板电脑的内存为 4GB、8GB 和 16GB，它的大小可由缓存大小控制。如果应用程序要使用比较大的 SQLite 数据库，需要设置该参数以保证设备在极限情况下仍能正常使用。

“VM 应用程序最大堆大小”设置了在系统终止 Android 应用程序前，Android 应用程序可以使用的堆大小上限。该选项会影响 SQLite 程序如何处理在堆上占用大量内存的对象。最有可能占用较多内存的对象就是大量类型为 cursor 的结果集。该选项的默认设置是 16MB。如果知道将要运行的 Android 应用程序设备内存的上限，那么就可以更改这个选项的设置，让应用程序在任何内存不足的情况下都能运行。

设备 RAM 的大小指定了设备的物理内存大小。这个选项不会成为 Android 应用程序的制约因素，但是为了能够在所有的设备上都能进行 SQLite 开发，需要调低该选项，以保证开发的设备尽可能接近未来的目标设备。默认设置是 96MB。

完成硬件选项的设置后，保存它，它将会出现在已有 Android 虚拟设备的列表中。

其他 Android 开发平台

对于那些已经熟悉 Android 开发或者需要跨平台开发工具的程序员，除了 Eclipse 外，还有其他开发工具可以进行 Android 开发或是基于 SQLite 的 Android 开发。尽量使用已经熟知的 Android 开发工具。

- PhoneGap：一个针对跨平台移动应用程序开发的开源开发框架，其网址为：www.phonegap.com。
- Appcelerator Titanium：另一个跨平台移动应用程序开发的开源框架，其网址为：www.appcelerator.com

以上两者都支持 Android 平台及其他开发平台下的 SQLite 开发。

285 Android SQLite 类和接口

现在已经搭建好了基于 SQLite 的 Android 应用程序开发环境。下一步探讨 Android 系统的类和接口，这些类和接口封装了 SQLite 底层的 C API。

不像其他的 API 接口，Android SQLite API 采用了一种不同的模式。

使用基础帮助类：SQLiteOpenHelper

Android 为 SQLite 数据库提供的第一个也是最重要的一个类就是 SQLiteOpenHelper 类，它的命名空间是 android.database.sqlite。这个类是可以供用户扩展的，在这个类里可以实现任何用户认为在创建、打开或使用数据库时的重要任务和行为。SQLiteOpenHelper 的构造函数很简单，具体如下：

```
SQLiteOpenHelper(Context context, String name,
                 SQLiteDatabase.CursorFactory factory, int version)
```

context 定义了应用程序运行的环境，包含应用程序所需的共享资源。name 定义了 Android 的数据库名字。factory 是一个新类，即 SQLiteDatabase.CursorFactory，这个类包含游标对象，存储查询 Android SQLite 数据库的结果集。version 指定了应用程序所用的数据库的版本。注意，不要和 SQLite 的真正版本混淆了。如果数据库不是参数 version 指定的版本，SQLiteOpenHelper 就会触发 onUpgrade 方法。SQLiteOpenHelper 类定义的所有方法如下：

```
synchronized void close()
synchronized SQLiteDatabase getReadableDatabase()
synchronized SQLiteDatabase getWritableDatabase()
abstract void onCreate(SQLiteDatabase db)
void onOpen(SQLiteDatabase db)
abstract void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
```

close()函数的含义一目了然，即关闭 SQLiteOpenHelper 对象中的 SQLite 数据库。

getReadableDatabase()和 getWriteableDatabase()行为相似，从它们的名字可以看出，它们只有一点不同。getReadableDatabase()以只读状态打开 SQLiteOpenHelper 对象中指定的数据库。也就是说，任何想要修改数据库的操作都是不允许的。getWriteableDatabase()也是打开数据库，但允许数据库正常的读/写操作。如果因为某种原因，数据库不能进行写操作，那么 getWriteableDatabase()将以只读方式打开数据库，并抛出一个类型为 SQLiteException 的异常。可以利用 isReadOnly()函数测试数据库是否是只读状态，如果数据库为只读状态，那么该函数返回 True，可以稍后使用 getWriteableDatabase()函数重新打开一个只读数据库，使它能够进行正常的读写。在一个不存在的数据库上调用任何方法时，都会隐式调用 helper 对象的 onCreat()方法。否则，如果是应用程序第一次访问一个数据库，会调用 onOpen()函数或者 onUpgrade()函数。默认会为应用程序缓存数据

集，直到 close()函数被调用时才释放缓存的数据库。

注意：打开一个大型数据库需要很长时间，特别是当 onOpen 或者 onCreate 函数中包含逻辑很复杂的操作时。我们强烈推荐在实际的应用程序中不要在程序启动时从应用程序的主线程调用这两个函数。

最后三个函数——onCreate()、onOpen()和 onUpgrade()在子类中是根据用户需要实现的。当数据库第一次被创建时调用 onCreate()函数，这个函数可以创建数据库表，并通过 insert 语句将数据加载到数据库。

onOpen()函数是当数据库被打开时调用的。应该在打开数据库时调用 isReadOnly()函数检查数据库的读写状态，以保证这个数据库的状态是已知的。

从应用程序的角度考虑，数据库需要升级时调用 onUpgrade()函数（注意，这里我们指的是应用程序的版本，而不是 SQLite 的版本）。一般而言，数据库的版本号从 1 开始，然后是 2、3，以此类推。您可以在该方法内编写自己的 alter table、create table、drop table 以及其他语句，升级的同时优雅地处理应用程序数据库模式的升级。

SQLiteDatabase 类

现在已经熟悉了 Android 系统中处理 SQLite 数据库相关的第一个类。接下来，将要介绍最关键的一个类 SQLiteDatabase 类。SQLiteDatabase 对象在概念上十分容易理解，与 SQLite C API 的底层数据库对象相似，但是实际上要复杂得多。

SQLiteDatabase 类中包括 50 个方法，每一个都有自己的用途，与其他方法之间也有细微差别。这些方法中的大部分方法都是完成一个简单的数据库任务，比如表的选择、插入、更新和删除语句。这里没有一一列出这些方法，只挑出一些重要的方法介绍，您可以利用业余时间了解其他方法。可以参考在线的 Android 文档了解 SQLiteDatabase 类。其网址为：http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html。

打开和关闭 SQLiteDatebase

SQLiteDatabase 提供了三种方法打开和关闭数据库，这些方法包括：

```
void close()
static SQLiteDatabase openDatabase(String path,
```

```
                                        SQLiteDatabase.CursorFactory factory, int flags)
static SQLiteDatabase openOrCreateDatabase(File file,
                                        SQLiteDatabase.CursorFactory factory)
static SQLiteDatabase openOrCreateDatabase(String path,
                                              SQLiteDatabase.CursorFactory factory)
```

close()方法就像它看起来那样简单。数据库关闭后，任何在 SQLiteDatabase 对象上 287
的查询都是不允许的。

openDatabase 函数是最常用的。参数 path 指定了 Android 应用程序中文件的位置，path 和关联的 factory 一起作为数据库的 CursorFactory。最后一个参数 flags 表明了打开数据库的方式。四个标志值定义了数据库打开的四种方式。

- OPEN_READWRITE：以只读/写方式打开数据库。
- OPEN_READONLY：以只读方式打开数据库。
- CREATE_IF_NECESSARY：如果数据库不存在，则先创建数据库。
- NO_LOCALIZED_COLLATORS：打开数据库，不支持 SQLite 本地化校对。存在一些匹配 Android 设备的地区设置的校对方式，以该方式打开时，可以无视这些校对方式。

第一个 openOrCreateDatabase 方法接受 File 对象作为参数，指出要打开的目标数据库。它没有 flags 参数，默认以 CREATE_IF_NECESSARY 方式打开。

第二个 openOrCreateDatabase 方法重载了 openDatabase 方法。它与 openDatabase 的功能相似，接收字符串路径作为参数指出数据库所在位置，它也没有 flags 参数，同样默认以 CREATE_IF_NECESSARY 方式打开数据库。

用 SQLiteDatebase 类执行普通查询

如同有很多创建 SQL 语句的方法，Android 也提供了大量的方法，可以在 SQLite 数据库中运行 SQL 语句。事实上，Android 提供了多达 16 种方法在 SQLite 数据库中运行一般的或特殊的查询。这些函数既包含 SQL 语句函数，比如单一表格的插入、更新等，也包括一般的执行 DML 和 DDL 的函数。这些函数分组显示如下：

```
void execSQL(String sql)
void execSQL(String sql, Object[] bindArgs)
Cursor query(boolean distinct, String table, String[] columns, String selection,
             String[] selectionArgs, String groupBy, String having, String orderBy,
             String limit)
```

```
Cursor query(String table, String[] columns, String selection, String[] selectionArgs,
             String groupBy, String having, String orderBy)
Cursor query(String table, String[] columns, String selection, String[] selectionArgs,
             String groupBy, String having, String orderBy, String limit)
Cursor queryWithFactory(SQLiteDatabase.CursorFactory cursorFactory, boolean distinct,
                        String table, String[] columns, String selection,
                        String[] selectionArgs, String groupBy, String having,
                        String orderBy, String limit)
Cursor rawQuery(String sql, String[] selectionArgs)
Cursor rawQueryWithFactory(SQLiteDatabase.CursorFactory cursorFactory, String sql,
                           String[] selectionArgs, String editTable)
```

不要被这么多的函数吓倒！它们可以归结为三个核心函数——execSQL()、query()和 rawQuery()，后两个函数有一个可选的参数 CursorFactory，这个参数是调用函数 SQLiteOpenHelper()的返回值。

两种格式的 execSQL()函数都有一个参数 sql，这个参数是一个 SQL 语句。第二个参数 bindArgs 接收一个数组，数组中的每个成员捆绑了一个查询。execSQL 函数运行那些没有返回值的查询语句，比如创建、插入、更新和修改表等。它的返回类型为 void 就说明了这点。

query()函数和 queryWithFactory()函数是在数据库中运行一些轻量级的单表查询语句的必备函数。观察它们的参数可以发现，变量包括 table、column、orderBy 等。本质上，这些方法允许将 SQL 语句传递给相关方法，而省去了 SQL 语句本身的关键字。

最后一组方法是 rawQuery()和 rawQueryWithFactory()，它们的参数是 SQL 查询语句，返回值是 Cursor 对象。这两个方法都有一个版本能够接收一个字符串数组 selectionArgs 作为参数。通过这个参数，SQLiteDatabase 对象将把捆绑的 SQL 语句中的问号（?）用这个数组中的值取代，按照一一对应的位置进行取代（也就是说，第一个问号（?）被数组中的第一个元素取代，以此类推）。

compilestatement 函数

SQLiteDatabase 中有一个方法我们没有讨论，这个方法是 compileStatement()。从它的名字看，您可能会认为它是一个很有用的可以用来调试查询语句的工具。从某些方面说，确实如此，但它有不同之处。首先，该函数的声明如下：

```
public SQLiteStatement compileStatement (String sql)
```

这个声明很普通，一个字符串的 SQL 语句作为参数，返回类型为 SQLiteStatement

对象。事实上，正如函数名所表示的，这个函数将编译查询语句并允许将其重用。这就有点奇怪了，与 compileStatement 函数关联的 SQLiteStatement 对象只能用于最多返回一行一列查询结果的查询。实际上，compileStatement 函数的返回结果也就是 SQLiteStatement 的数据成员必须是简单的标量值。让我们想象这可以应用于哪种情况，比如对表中的行计数或者对结果集计数。

现在已经了解了 compileStatement()函数和 SQLiteStatement 对象，但是如果您像我们一样的话，就会很少用到它。

使用 SQLiteDatabase 的快捷方法

我们提到 SQLiteDatabase 中一共有 16 种方法，但上文中只介绍了 8 个函数。另外 8 个函数允许在数据库表中进行插入、更新、删除和替换操作。这些方法如下：

```
int delete(String table, String whereClause, String[] whereArgs)
long insert(String table, String nullColumnHack, ContentValues initialValues)
long insertOrThrow(String table, String nullColumnHack, ContentValues initialValues)
long insertWithOnConflict(String table, String nullColumnHack,
                          ContentValues initialValues, int conflictAlgorithm)
long replace(String table, String nullColumnHack, ContentValues initialValues)
long replaceOrThrow(String table, String nullColumnHack, ContentValues initialValues)
int update(String table, ContentValues values, String whereClause, String[] whereArgs)
int updateWithOnConflict(String table, ContentValues values, String whereClause,
                         String[] whereArgs, int conflictAlgorithm)
```

289

上面这些函数都很容易理解。不同的操作，其参数列表也不相同。具体了解 insert()函数和与它有关联的 replace()函数时，您会发现字符串参数 nullColumnHack 有点奇怪。这个参数与 initialValues 和一个 ContentValue 相关联映射。如果 initialValues 的值为空，那么 nullColumnHack 将会为列提供一个 NULL 值，以防止插入一个没有数据的条目而引起失败。

尽管这些方法看上去很吸引人，但是它们主要是为了吸引那些面向对象的程序员设计的，这些程序员更愿意使用 CRUD 类型的基本变量进行操作。我们已经讨论了一些函数，抛砖引玉，相信您已经可以很好地利用这些函数构建和执行查询了。

用 SQLiteDatabase 类管理事务

Android 很看重 SQLite 事务管理。Android 系统中有很多方法是用来启动、结束和

管理事务的。

```
void     beginTransaction()
void     beginTransactionWithListener(SQLiteTransactionListener transactionListener)
void     endTransaction()
boolean      inTransaction()
void     setTransactionSuccessful()
```

上述大部分方法无须解释。beginTransaction()启动 SQLite 事务，endTransaction()结束当前的事务。重要的是，决定事务是否被提交或者回滚取决于事务是否被标注了“clean”。setTransactionSuccessful()函数用来设置这个标志位，这个额外的步骤刚开始让人反感，但事实上它保证了在事务提交前对所做更改的检查。如果不存在事务或者事务已经是成功状态，那么 setTransactionSuccessful()函数就会抛出 IllegalStateException 异常。inTransaction()函数测试是否存在活动事务，如果有，则返回 true。

现在就剩下 beginTransactionWithListener()函数了。该函数的参数是 SQLiteTransaction-Listener 对象，SQLiteTransactionListener 负责监听与事务相关的一切操作，比如提交、回滚或者嵌套地开始一个新事务。

使用其他 SQLiteDatabase 函数

SQLiteDatabase 对象还有很多其他有用的方法。下面选择其中一些进行讨论。

- public long getMaximumSize()：返回数据库允许的最大值。
- public int getVersion()：返回应用程序所用的数据库的版本号。
- public boolean isDbLockedByCurrentThread()：测试当前线程是否锁住了数据库。
- public boolean isDbLockedByOtherThreads()：测试是否有其他线程锁住了数据库。
- public static int releaseMemory()：释放数据库不再需要的资源，比如内存等。返回释放资源的字节数。

还有很多简单的测试方法测试数据库方方面面的状态，例如，有一些函数是用来控制将缓存中的数据库副本同步到存储器中等。

在实际中应用 SQLiteOpenHelper 和 SQLiteDatabase

现在已经介绍了足够多的 Android SQLite API 方法，下面将利用这些方法继承 SQLiteOpenHelper 类，重载类中的方法，管理我们底层的 SQLite 数据库 SQLiteDatabase。

代码 10-1 是一段简单的 Android 代码片段，它实现了自己的 SQLiteOpenHelper 子类，处理自己的 Seinfeld 数据库常用的创建和打开操作。这段代码包含在实例代码的 myDatabaseHelper.java 文件中。

代码 10-1　myDatabaseHelper.java

```
import java.io.*;
import android.database.sqlite.*;
import android.database.SQLException;
import android.content.Context;

public class myDatabaseHelper extends SQLiteOpenHelper{
  //对于给定的应用程序，DBPATH 使用默认系统路径
  // /data/data/<app namespace> ，在本示例中是 com.example.seinfeld
  private static String DBPATH = "/data/data/com.example.seinfeld/databases/";
  private static String DBNAME = "foods.db";
  private SQLiteDatabase myDatabase;
  private final Context myContext;

  //构造函数
  public myDatabaseHelper(Context context) {
    super(context, DBNAME, null, 1);
    this.myContext = context;
  }

  //创建一个空的 db，然后替换所选的 db
  public void createDatabase() throws IOException{
    if (!checkDatabase()) {
      this.getWritableDatabase();
      try {
        copyDatabase();
      }
      catch (IOException e) {
        throw new Error("Error copying database from system assets");
      }
    }
  }

  //检查数据库是否已存在
  private boolean checkDatabase(){
    SQLiteDatabase checkableDatabase = null;
```

```
    try {
      checkableDatabase =
        SQLiteDatabase.openDatabase(DBPATH+DBNAME, null,
                                    SQLiteDatabase.OPEN_READONLY);
    }
    catch (SQLiteException e) {
      //因为数据库不存在，所以我们返回 false.
    }
    if (checkableDatabase != null) {
      checkableDatabase.close();
    }
    return checkableDatabase != null ? true : false;
  }

  //从应用程序资产中复制数据库
  //越过使用空数据库
  private void copyDatabase() throws IOException{

    InputStream myInput = myContext.getAssets().open(DBNAME);
    OutputStream myOutput = new FileOutputStream(DBPATH+DBNAME);
    byte[] buffer = new byte[1024];
    int length;
    while ((length = myInput.read(buffer))>0){
      myOutput.write(buffer, 0, length);
    }
292 myOutput.flush();
    myOutput.close();
    myInput.close();
  }

  public void openDatabase() throws SQLException{
    myDatabase = SQLiteDatabase.openDatabase(DBPATH+DBNAME, null,
                                             SQLiteDatabase.OPEN_READWRITE);
  }


  @Override
  public synchronized void close() {
    if(myDatabase != null)
      myDatabase.close();
```

```
    super.close();
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    //Handle creation tasks, etc.
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    //Handle upgrade tasks, etc.
  }
} //结束 myDatabaseHelper.java
```

现在可以使用 myDatabaseHelper 类打开或者创建 Seinfeld Food 数据库了。

```
…

myDatabaseHelper mySeinfeldDBHelper = new myDatabaseHelper();
mySeinfeldDBHelper = new myDatabaseHelper(this);

try {
  mySeinfeldDBHelper.createDatabase();
}
catch (IOException e) {
  throw new Error("Failed to create Seinfeld database");
}

try {
  mySeinfeldDBHelper.openDatabase();
}
catch(SQLException e) {
  throw e;
}
…
```

本章从头至尾都要使用 myDatabaseHelper 类构建基于 SQLite 的 Android 应用程序。

使用 SQLiteQueryBuilder 类查询 SQLite 数据库

293

Android 系统中的 SQLite 类包既支持数据库格式的数据库查询，也支持语句格式的

数据库查询。我们已经介绍过 SQLiteDatabase 对象，现在介绍另一个类——SQLite-QueryBuilder。

与 SQLiteDatabase 类类似，SQLiteQueryBuilder 提供了很多帮助创建 SQL 语句的方法，使用这些方法时不必在语句中提供 SQL 语句的保留字。它与 SQLiteDatabase 的主要不同点在于，SQLiteQueryBuild 能够直接为 SQLiteDatabase 对象上的语句提供功能，并且能够将字符串传递给 SQLiteDatabase 对象。首先，认识能够支持直接执行语句的方法：

```
Cursor query(SQLiteDatabase db, String[] projectionIn, String selection,
             String[] selectionArgs, String groupBy, String having, String sortOrder)
Cursor query(SQLiteDatabase db, String[] projectionIn, String selection,
             String[] selectionArgs, String groupBy, String having, String sortOrder,
             String limit)
```

这些方法看起来十分相似。它们与 SQLiteDatabase 类中的 query 方法很像，只有几点不同。这里的方法都是将数据库对象作为第一个参数，而不是直接在数据库对象上调用方法。很多人会认为这只是个人偏好问题，但在很多情况下，SQLiteQueryBuilder 的这种方法显得更有益处。您会发现这两个函数之间的不同在于第二个函数里多了一个 limit 参数。

细心的读者还会发现另一个不同之处：table 参数去哪里了？这是 SQLiteQueryBuilder 与 SQLiteDatabase 类另一个主要的不同点。有一个单独的函数 setTables()用来指定对哪个数据库表进行查询操作。

```
public void setTables (String inTables)
```

> **注意**：这是程序员在处理 SQLiteQueryBuilder 类和 SQLiteDatabase 类之间的转换时最常遇到的问题。

setTables 方法用一个被逗号分隔的表名作为参数。SQLiteQueryBuilder 类中还有一些其他方法的机制也与 setTables 方法类似，包括：setDistinct 方法，该方法指出 SQL 语句应该使用 distinct 关键字；setProjectionMay(Map<String, String> columnMap)方法，该方法控制列的别名，消除了列别名模棱两可的情况。

SQLiteQueryBuilder 类中还有很大一部分方法是“构建”函数。这些方法构建一个查询字符串并返回，返回值可以用于 SQLiteDatabase 对象或递归使用在 SQLiteQueryBuilder

对象中。

```
String buildQuery(String[] projectionIn, String selection, String[] selectionArgs,
                  String groupBy, String having, String sortOrder, String limit)
static String buildQueryString(boolean distinct, String tables, String[] columns,
                               String where, String groupBy, String having,
                               String orderBy, String limit)
String buildUnionQuery(String[] subQueries, String sortOrder, String limit)
String buildUnionSubQuery(String typeDiscriminatorColumn, String[] unionColumns,
                          Set<String> columnsPresentInTable int computedColumns
                          Offset,
                          String typeDiscriminatorValue, String selection,
                          String[] selectionArgs,String groupBy, String having)
```

294

同样，这些方法与 SQLiteDatabase 中功能相同的方法很相似。但是，这些方法更加关注联合查询。对作者而言，发现自己创建联合查询语句，然后使用 SQLiteDatabase 的 rawQuery 方法更加容易，但是如果利用编程实现联合查询语句，buildUnionQuery()以及其他相关的方法对 SQL 注入攻击有更好的防范性。

Android 的 SQLite 类库背后具有丰富的理论支持。现在我们来创建基于 SQLite 的 Android 应用程序。

搭建 Seinfeld Android SQLite 应用程序

在搭建好工作的开发环境并掌握了 Android SQLite 类库的相关知识后，就可以开始搭建自己的基于 SQLite 的 Android 应用程序了。我们的需求与第 9 章中在 iOS 系统中搭建应用程序需求是类似的：

- 建立一个 Android 应用程序，按照受欢迎的程度显示 Seinfeld 数据库中的食物。
- 对于任何一种受欢迎的食物，显示它在整个 Seinfeld 数据库中被提到的次数。
- 当食物是第一次出现时要给出提示。

明确了简单需求后，我们将使用在本章前面创建的 myDatabaseHelper 类和 Android 提供的 UI 实现这个应用程序。在介绍时会涉及一些与 SQLite 无关的领域知识，我们会尽量少地用到它们。

下面是要遵循的一些步骤：

1．在 Eclipse 中创建一个新的 Android 工程。

2．将 SQLite 框架添加到这个工程。

3．将 Seinfeld foods.db 添加到应用程序的资源库。

4．定义类，并将 SQLite 数据库中的 food 数据映射到代码。

5．定义 UI 元素，显示 SQLite 中数据的详细和概要信息。

6．定义 UI 的行为。

7．启动！

现在开始。

295 创建新的 Android 工程

启动 Eclipse，选择“File”→“New”→“Android project”，会弹出新建 Android 工程的对话框，如图 10-4 所示。

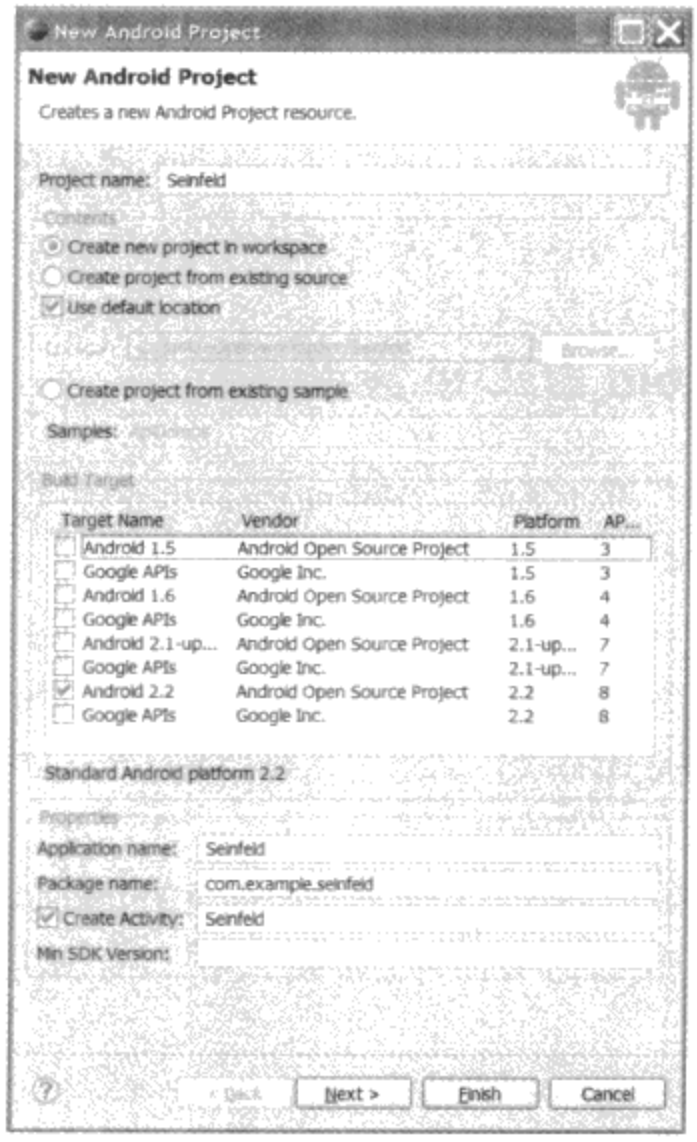


图 10-4　创建新的 Android SQLite 工程

填写工程名、应用程序名和包的名字，如图 10-4[①]所示，或者使用容易记住的有意义的名字。应用程序名将作为应用程序的标题出现，包的名称要遵循 java 包的命名规则——名字需在全局范围内唯一。 296

将 Seinfeld SQLite 数据库添加到工程

打开 Seinfeld 工程文件夹，然后打开资源文件夹，将 SQLite 数据库文件 foods.db 添加到工程中。默认情况下，资源文件夹是空的，可以将 foods.db 文件从操作系统拖动到资源文件夹里，或者选择将数据库文件链接到 foods.db 文件上。我们不推荐第二种方法，因为这样做会加大应用程序打包的难度。

我们将在本章后面的部分介绍如何利用实用工具 adb 在数据库文件中使用命令行。

查询 Food 数据库表

我们需要在 Seinfeld 应用程序中添加额外的函数，查询数据库，返回我们已知并喜爱的食物。我们还需要将查询结果打包传递给 Android UI 中的 ListView，这样就能通过 Android 应用程序查看和应用这些结果。

第一个任务就是在 myDatabaseHelper 类中添加一个方法获得 Food 数据库表中所有食物的列表。下面是节选自 myDatabaseHelper.java 中示例代码的片段：

```
…
public Cursor fetchAllFoods() {

  return myDatabase.rawQuery("select name from foods order by name", null);
}
…
```

这段代码并不复杂。我们可以在这段代码的基础上增加代码，以便完成更复杂的 Seinfeld 应用程序。

接下来，需要在 ListView 中为所有返回的食物添加一个游标来显示它们。在 Seinfeld.java 代码中，声明了很多字符串代表已知域的名字，并设定了一个函数遍历结果集，并将域与结果集中的值绑定，赋值给 ListView：

① 原文为图 10-3，可能有误，故翻译时修改为图 10-4。——译者注

```
...
  public static final String FOODS_NAME_FIELD = "name";
...

private void fillData() {
  Cursor foodCursor = mDbHelper.fetchAllNotes();
  startManagingCursor(foodCursor);

  // 为食物名称创建一个数组
  String[] from = new String[]{myDatabaseHelper.FOODS_NAME_FIELD};

297  // 创建一个域数组用于绑定
  int[] to = new int[]{R.id.text1};

  SimpleCursorAdapter foods =
    new SimpleCursorAdapter(this, R.layout.food_row, foodCursor, from, to);
  setListAdapter(foods);
}
```

不用担心，如果 Android 特定的 UI 调整没有意义，接下来我们将讨论神秘的 R 对象以及 text1 域是如何定义的。

定义用户接口

刷新刚刚创建的 Seinfeld 工程时，一个空白的画布会出现在用户接口的位置。如果仔细观察工程，会发现在 res 文件夹下有一个 main.xml 文件，这个文件控制着 UI 设计，可以通过图形化接口建造器或者手工修改 Android UI XML 来设计 UI。

下面还是通过例子来讲解如何定义用户接口。代码 10-2 显示了 food_list.xml 文件的内容，这个文件控制了应用程序整体的风格和样式。

代码 10-2　food_list.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

        <ListView android:id="@id/android:list"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
```

```
        <TextView android:id="@id/android:empty"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/no_foods"/>
</LinearLayout>
```

简而言之，我们的设计遵从 LinearLayout 规范，这也是文本应用程序最常用的规范。在里面，我们为 UI 定义了一组预期的行为规范，指定了哪些 UI 会被显示。我们尝试用列表语法显示 ListView。在 TextView 中显示字符串 no_foods，或者显示实际的文本内容。在同一时间里，这两个视图只会有一个被使用。list 和 empty 这两个值是 Android 平台提供的，目的是为了降低定义 UI 组件行为的复杂性。

配合另一个视图文件 food_row.xml（见代码 10-3）一起看，会更加一目了然。

代码 10-3　food_row.xml

298

```
<?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/text1"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

这里我们创建了一个新的名为 text1 的 ID。符号+表示如果忘记定义 text1，在这里定义它，以便马上使用它。这种方法十分便捷，可以确保在使用 UI 元素时已经定义了该元素。

将 UI XML 文件包含在工程中。通过强大的 Android 开发工具，Eclipse 会注意到新定义域和 ID，并添加必需的变量到神秘的 R.java 文件中。实际上，这个文件并没有那么神秘。R.java 文件是由 ADT 管理的，自动连接到 Java 代码中的 UI 资源，可以查看 R.java 文件的内容，但是基本不需要手动修改它。

连接数据和用户接口

最后一项任务就是何时怎样控制接口接收数据。为了实现这一功能，需要扩展之前实现的 onCreate 函数，实现查找和打开 SQLite 数据库 food，并将 fillData()返回的结果传递给 ListView 进行显示。这段代码包含在 Seinfeld.java 文件中。

```
…
    @Override
```

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.food_list);
        mySeinfeldDbHelper = new MyDatabaseAdapter(this);
        mySeinfeldDbHelper.createDatabase();
        mySeinfeldDbHelper.openDatabase();
        fillData();
    }
…
```

这里，调用 setContentView 函数接收 ListView 和 R.layout.food_list。下一步创建一个 myDatabaseAdapter 类对象，创建（复制到缓存）并打开数据库。接下来调用 fillData 函数，在之前的定义中，该函数接收数据库中返回的数据，并把它们映射到 TextView 元素中，在 ListView 中显示。

仔细阅读 Seinfeld.java 文件中的代码和示例文件中相关的代码。本书就不浪费大量的纸张打印那些很容易看明白的代码，但是阅读全部的代码将有助于加深理解。

299

查看完成的 Seinfeld 应用程序

我们已经建立好了 Seinfeld 应用程序，现在看看它是怎样工作的吧。在 Eclipse 中，选择“Run”→“Run As”→“Android Application”，这样会在 Android 虚拟设备中启动应用程序。

■**注意**：虚拟设备的初始化、启动、加载过程需要几分钟。之后，只需要几秒钟就能看见应用程序了，还要保证已经解开了屏幕保护锁，以保证 Seinfeld 应用程序耐心地在后台等待。

Seinfeld 应用程序启动后，可以看到 SQLite 数据库 foods.db 中熟悉的食物名称，如图 10-5 所示。

300

好了，现在已经完成了第一个 Android SQLite 应用程序。从现在起，您可以任意添加新的功能和设计新的应用。

SQLite Android 应用程序的注意事项

有些运行 Android 系统的平台可能对硬件和资源有些限制（与一般的桌面计算机和

笔记本电脑相比），所以，在开发 Android 应用程序和设计用户界面时都要格外注意硬件和资源条件的限制，确保用户获得良好的体验。

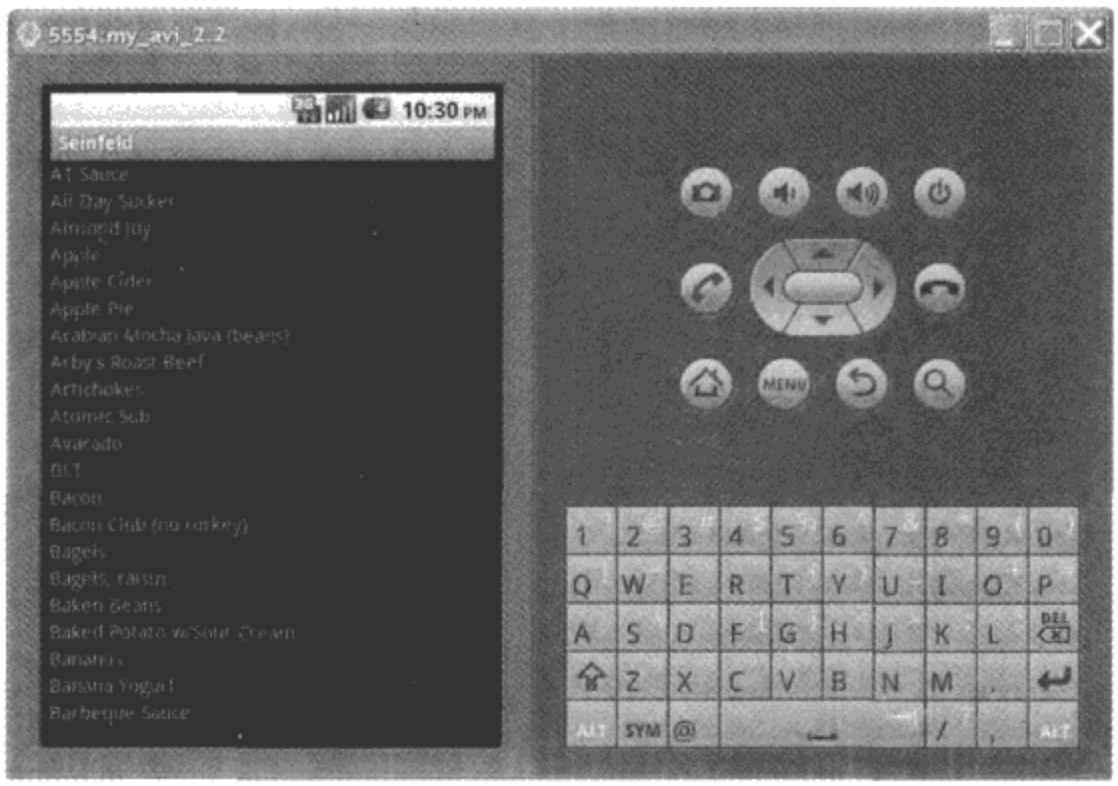


图 10-5　运行的 Seinfeld Android SQLite 应用程序

Android 数据库的备份

观察 myDatabaseHelper.java 文件中子类的定义可以看出，可以直接对 SQLite 的数据库文件进行操作。使用一般的 Android API 调用，可以自己控制 SQLite 文件，因此，可以使用这种方法为数据库提供一个备份和恢复，但是有一种更好的方法。

Google 提供了 Android 的备份服务，允许将任何形式的数据复制到远程存储设备中。这项服务不是只针对 SQLite 数据库文件，不过 SQLite 是我们今天介绍的重点。这是一个十分有效的数据存储、备份和恢复方法，比任何人工备份的效果都要好。例如，如果用户触发了 Android 设备上的复位键，用户自己的备份很可能丢失。但当您重新安装并应用时，数据备份系统可以自动恢复您的数据。

这项服务的功能很全面，是基于很多类实现的。其中主要的 API 是 BackupAgent、BackupAgentHelper 和 BackupManager。有些类我们已经见到了，这些类都包含很多方法和子类，可以重载这些类，以提供在备份和恢复时需要执行的行为。有很多资料介绍该备份功能，我们推荐在 http://developer.android.com/guide/topics/data/backup.html 中查

看在线文档。

Android 系统处理大型 SQLite 数据库

在介绍搭建 Android 虚拟设备时，我们提到了一些配置参数来控制和模拟在创建 Android 应用程序时的内存、缓存、硬盘大小，这些参数包括应用程序最大虚拟内存堆大小、缓存区大小和设备 RAM 大小。

当处理 SQLite 数据库中的大数据集时会遇到一个棘手问题。运行一些查询，返回一个很大的结果集游标对象，开始处理这些数据，突然 UI 崩溃，在堆栈里输出了一些奇怪的 UI 错误。这时怎么处理？

尽管现在 Android 设备和 AVD 上配置的内存和硬盘已经相当大了，但是还有一些其他对设计的限制。最常见的这种限制包括使用 CursorWindow 对象作为游标结果集的二级缓存。CursorWindow 只能容纳 1MB 数据。您所见到的各种各样的程序崩溃现象都是由于数据包含了过多的行或列，包含了过长的数据，超过了 CursorWindow 大小的限制。当然，有时数据很短，所以运行时没有问题，但是，有些时候就崩溃了！

我们的建议是要时刻记住：在那些只有 3 英寸到 4 英寸大小的屏幕上显示千行是不现实的。哪些用户才会在一次操作中滚动或读取如此大量的数据，特别是在这样小屏幕的设备上？

总结

现在已经了解了创建基于 SQLite 的 Android 应用程序的工具、类和实施方法。Android SQLite API 与其他的 API 不同，正如 Android 平台更新的速度一样，Android API 更新速度也十分快。

因为 Android 系统已经被证明是一个十分成功的平台，我们坚信您为 Android 系统开发基于 SQLite 的应用程序也会非常成功。祝您好运！

第 11 章

SQLite 内部机制及新特性

本章介绍 SQLite 内部机制的有关内容。这里收集了 SQLite 开创者 Richard Hipp 在过去一年所提出的相关论述，特别是 SQLite 最新版本中一个较大的变化：SQLite 预写（Write-Ahead）日志模型。也会简要介绍 SQLite 中 pager 和 B-tree 的运行机制，并比较详细地介绍 SQLite 的基本数据类型、亲缘性以及类型处理方法。

由于SQLite网站已经详细地介绍了SQLite的内部工作机制，本书也不宜追溯VDBE中的命令、优化模型等有关 SQLite 健康性能指标的变化。所以，在此就不做 SQLite 内部工作机制的详尽介绍。希望您能喜欢这些内容，并多多阅读，以增强对 SQLite 优雅设计的领悟。

B-tree 和 Pager 模块

B-tree 为 SQLite VDBE 提供 $O(\log N)$级时间复杂度的查询和插入，通过双向遍历记录实现 $O(1)$级时间复杂度的删除。B-tree 是自平衡的，并能够对碎片清理和内存再分配进行自动管理。B-tree 对如何读写磁盘没有限定，只是关注页之间的关系。B-tree 在需要一个页或修改一个页时，会通知 pager。修改页时，如果使用传统的回滚日志，pager 首先将原始页复制到日志文件。同样，pager 在 B-tree 完成写操作时收到通知，并基于所处的事务状态决定如何处理。

数据库文件格式

数据库中所有的页都是以 1 开始顺序编号的。一个数据库是由多个 B-tree 组成的——每张表以及每个索引各对应一个 B-tree（表使用 B+tree；索引使用 B-tree）。数据库中每

张表或索引都以根页面作为第一页。所有的索引和表的根页面都存储在 sqlite_master 表中。

304 数据库的第 1 页比较特殊。第 1 页的前 100B 包含了文件头，用于说明数据库文件，它包含了诸如这样的信息：库版本、模式版本、页大小、编码方式、是否启用自动清理——所有创建数据库所需的数据库持久性设置以及通过编译指示设置的其他参数。文件头的详细内容是在 btree.c 文件中定义的，可通过 http://www.sqlite.org/fileformat2.html 查看。同时，第 1 页还是 sqlite_master 表的根页面。

页重用及清理

SQLite 使用空闲链表回收页，即，当一个页上的所有记录被删除时，该页将被存回至空闲页链表中预留重用。之后，若需添加新信息，最近回收的页将被选中，以避免创建新页（创建新页需要增加数据库文件的大小）。运行 VACUUM 命令可以释放空闲链表中的页并因此减小数据库占用的空间。实际上，VACUUM 命令重新建立新的数据库文件并将所有使用的页复制到新文件中，同时忽略空闲链表中的页，从而得到一个新的压缩的数据库文件。如果启用数据库空闲页自动清理功能，SQLite 将不会使用空闲页链表，而是在每个提交行为时自动进行数据库的收缩处理。

B-tree 记录

B-tree 中的页由一系列 B-tree 记录组成，这些记录也称为有效载荷。这些记录并非你想象的数据库记录——表中有多个列的那种格式，而是更为原始的格式。一个 B-tree 记录（有效载荷）仅由两个域组成：键值域和数据域。键值域是每个数据库表中所包含的 ROWID 值或主键值；在 B-tree 中，数据域可以包含任意类型的内容。最终，数据库记录信息存储在数据域中。B-tree 用来保持记录有序并方便记录的查询，同时，键值域能够完成 B-tree 的主要工作（但例外的是，定位下一个记录由 B+tree 负责）。此外，记录（有效载荷）的大小是可变的，这取决于内部键值域和数据域的大小。一般而言，每个页拥有多个有效载荷，但如果有效载荷超出了一个页的大小，将会出现跨越多个页的情况（如包含 blob 类型的记录）。

B+tree

B-tree 记录按键值序存储。所有的键值在一个 B-tree 中必须是唯一的（由于键值对应于 rowid 主键，并且 SQLite 很重视该域，所以唯一性是可以保证的）。表使用 B+tree

定义在内部页中，不包含表数据（数据库记录）。图 11-1 为 B+tree 表示一个表的示例。

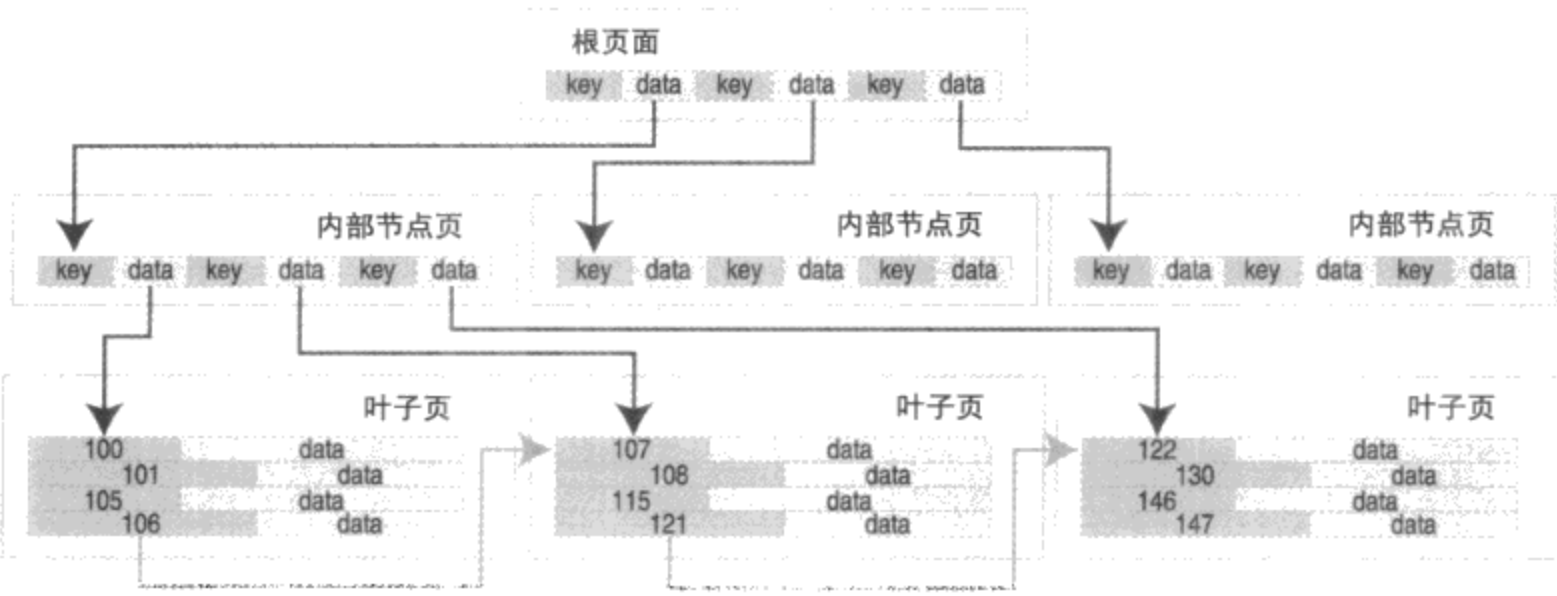


图 11-1　B+tree 组织结构（表）

B+tree 的根页面和内部节点页都用于搜索导航。这些页中的数据域均指向下一层页，这些页只包含键值。所有的数据库记录都存储在叶子页中，在叶子页层，记录和页按键值序排列，以便 B-tree 游标能够遍历记录（水平遍历）。无论是前向访问还是后向访问，只需使用叶子页，从而使得在 $O(1)$时间复杂度下遍历记录成为可能。

记录和域

叶子页的数据域中的数据库记录由 VDBE 控制。数据库记录以特殊的二进制格式存储，这些格式描述了记录中的所有域。记录格式由连续字节流形式的逻辑头段和数据段组成。逻辑头段包括逻辑头大小（由可变大小的 64b 整数表示）和用于描述数据段中每个存储域类型（同样也是可变大小的 64b 整数）的数组，如图 11-2 所示。可变大小的 64b 整数采用 Huffman 编码实现。

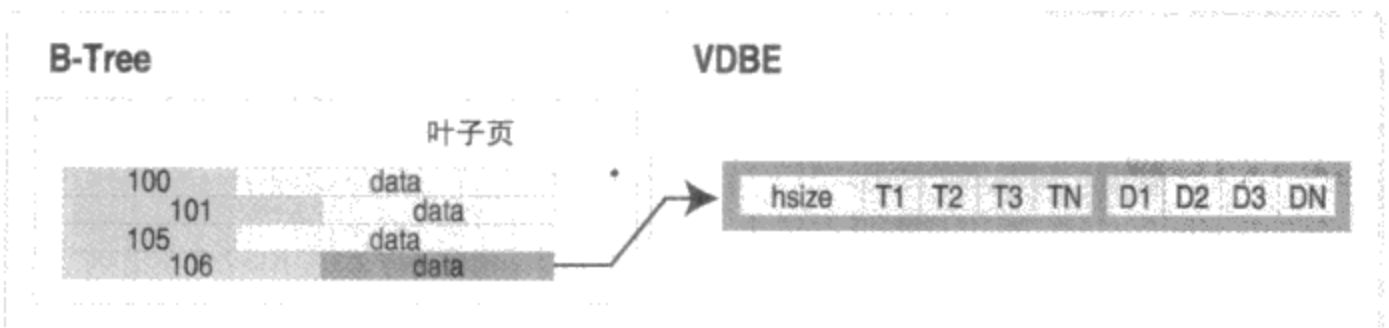


图 11-2　记录结构

类型个数与记录域个数一致。此外，类型数组中的每个索引对应于域数组的索引。类型描述了对应域值的数据类型和大小。表 11-1 列出了可能的类型及其含义。

306

表 11-1　域类型值

类 型 值	含　义	数据长度
0	NULL	0
N in 1..4	有符号整数	*N*
5	有符号整数	6
6	有符号整数	8
7 IEEE	浮点	8
8	整数常量 0	0
9	整数常量 1	0
10、11	预留将来使用	N/A
N > 12 并且是偶数	blob	(*N*-12)/2
N>13 并且是奇数	文本	(*N*-13)/2

例如，episodes 表中的第一个记录：

```
Sqlite> SELECT * FROM episodes ORDER BY id LIMIT 1;
id    season   name
---   ------   --------------------
0     1        Good News Bad News
```

图 11-3 显示了该记录的内部记录格式。

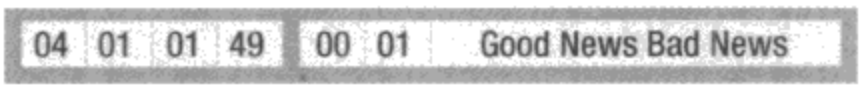


图 11-3　episodes 表的第一个记录

记录头的长度是 4B，从头的大小可以看出，其本身采用单字节编码。第一个类型对应 id 域，为 1B 的有符号整数。第二个类型对应 season 域，同样也为 1B 的有符号整数。name 域类型标识是一个奇数，表明该域数据为文本（text）类型值。name 域大小为(49-13)/2=18B。通过这些信息，VDBE 能够解析记录的数据段并提取各字段数据。

307

层次化数据组织

基本上，具体的数据单元由堆栈中的各模块处理。自下而上，数据变得更精确、详

细。自上而下，数据变得更集中、模糊。具体地说，C API 负责处理域值，VDBE 负责处理记录，B-tree 负责处理键值和数据，pager 负责处理页，操作系统接口负责处理二进制数据和原始数据存储，如图 11-4 所示。

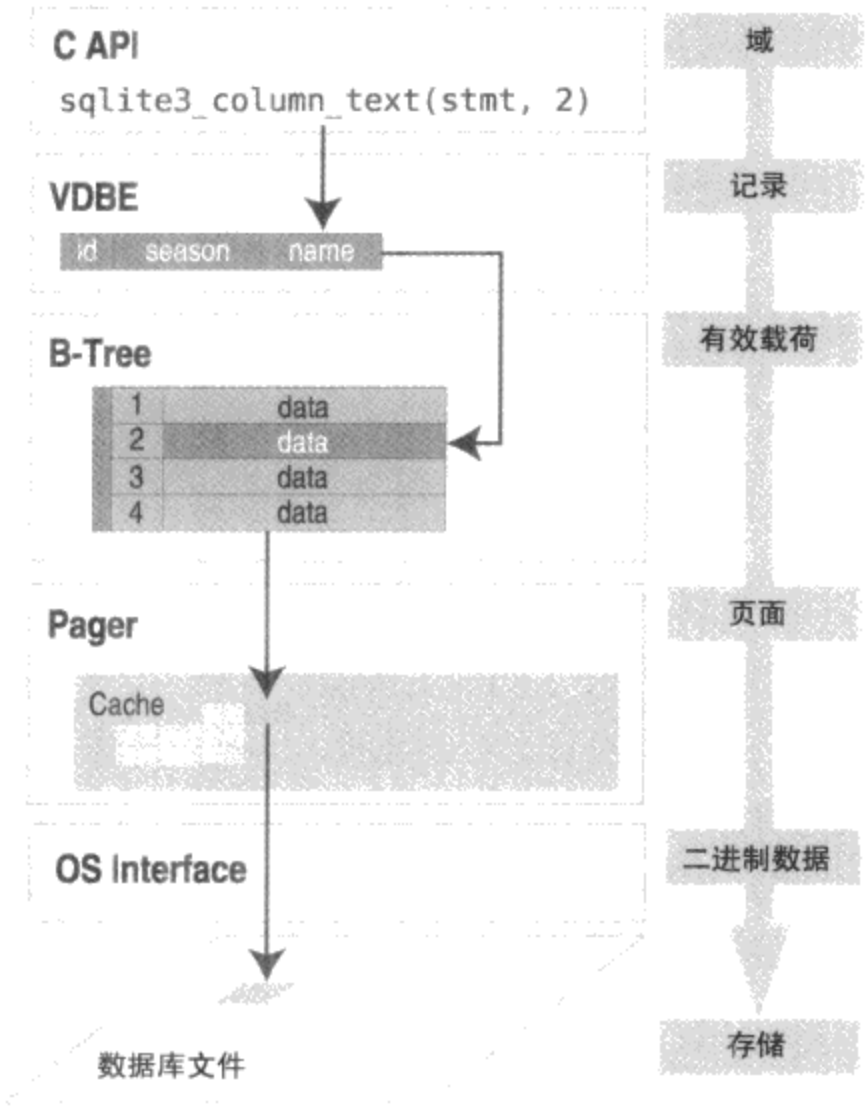


图 11-4　模块和其他相关数据

每个模块负责维护自身在数据库中对应的数据部分，然后依靠底层提供所需提取信息的初始数据，并从中提取需要的内容。

页面溢出

如前所述，有效载荷及其内容可有不同的大小。然而，页面大小是固定不变的。因此，给定的有效载荷总有可能超出单页装载大小。这种情况发生时，额外的有效载荷将添加到溢出页面的链接链表上。由此看来，有效载荷将在有序的链接链表中显示，如图 11-5 所示。

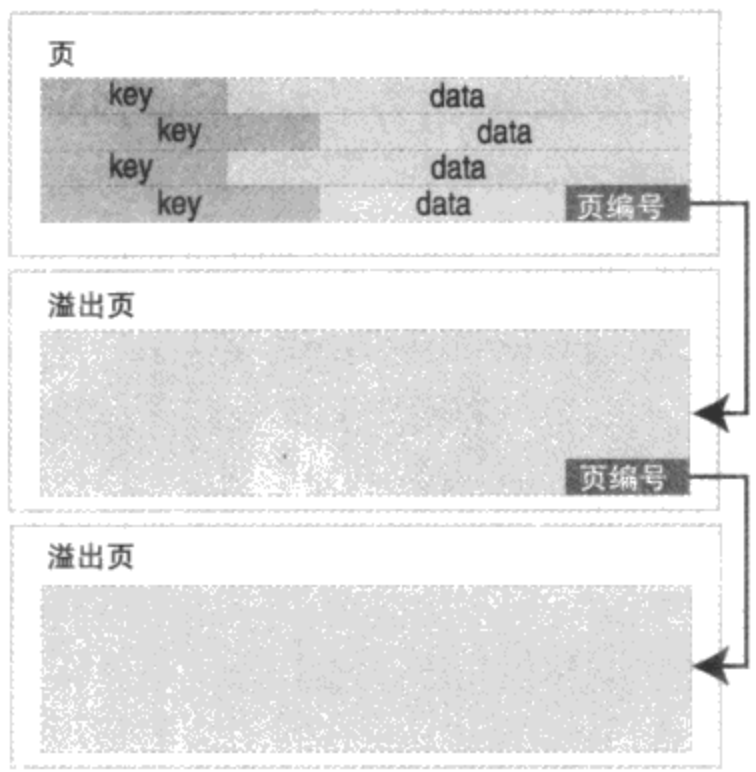


图 11-5　页面溢出

图 11-5 中第 4 个有效载荷超出了当前页所能装载的大小。因此，B-tree 模块创建了溢出页来装载。实际上，一个溢出页也不能装载，因此，又链接了第二个溢出页。这实际上就是处理二进制大对象的方法。使用真正的大字段时，最后都采用页链接链表来存储。如果 blob 字段太大，这种方式效率很低，此时，可考虑创建外部文件来存储 blob 数据，并将外部文件名保存在记录中。

B--tree API

B-tree 模块有自己的 API，它们是独立于 C API 的。本节研究 B-tree API 的目的是比较全面地了解 SQLite 对 B-tree 的使用，但并不意味着它们是 B-tree 的通用 API。这些 API 主要是针对 SQLite 需求量身定做的，因此，不宜直接应用于其他的项目。换言之，比较深刻地理解这些 API 后，更有助于了解 SQLite 的内部机制。B-tree 模块的另一个优点是本身支持事务机制、pager 处理事务、锁以及日志文件，并为 B-tree 模块提供支持。B-tree API 根据目的大体划分成以下函数。

存取及事务函数

数据库及事务例程如下：

- sqlite3BtreeOpen：打开一个新数据库文件并返回 B-tree 对象。
- sqlite3BtreeClose：关闭数据库。
- sqlite3BtreeBeginTrans：开始一个新事务。
- sqlite3BtreeCommit：提交当前事务。
- sqlite3BtreeRollback：回滚当前事务。

309

- sqlite3BtreeBeginStmt：开始一个语句事务。
- sqlite3BtreeCommitStmt：提交一个语句事务。
- sqlite3BtreeRollbackStmt：回滚一个语句事务。

表函数

表管理例程如下：

- sqlite3BtreeCreateTable：在数据库文件中新建一个空 B-tree，是采用表格式（B+ tree）还是索引格式（B-tree），由符号设置决定。
- sqlite3BtreeDropTable：删除数据库中的一个 B-tree。
- sqlite3BtreeClearTable：删除 B-tree 中所有的数据，但保持 B-tree 结构完整。

游标函数

游标处理函数如下：

- sqlite3BtreeCursor：创建一个指向特定 B-tree 的游标。游标可以是读游标，也可以是写游标，但是读游标和写游标不能同时在同一个 B-tree 中存在。
- sqlite3BtreeCloseCursor：关闭 B-tree 游标。
- sqlite3BtreeFirst：移动游标至 B-tree 第一条记录。
- sqlite3BtreeLast：移动游标至 B-tree 最后一条记录。
- sqlite3BtreeNext：移动游标至当前游标所指记录的下一条。
- sqlite3BtreePrevious：移动游标至当前游标所指记录的前一条。
- sqlite3BtreeMoveto：移动游标至同指定键值匹配的一条记录。如果没有匹配的记录，则移动游标至一条已存在并接近匹配记录的位置。

记录函数

310

关键字和记录函数如下：

- sqlite3BtreeDelete：删除游标所指记录。

- sqlite3BtreeInsert：在 B-tree 的适当位置插入一条记录。
- sqlite3BtreeKeySize：返回当前游标所指记录的关键字长度（字节）。
- sqlite3BtreeKey：返回当前游标所指记录的关键字。
- sqlite3BtreeDataSize：返回当前游标所指记录的数据长度（字节）。
- sqlite3BtreeData：返回当前游标所指记录的数据。

配置管理函数

设置各种参数变量值的函数如下：

- sqlite3BtreeSetCacheSize：控制页缓存大小以及同步写入（在编译指示 synchronous 中定义）。
- sqlite3BtreeSetSafetyLevel：改变磁盘数据的同步方式，以增加或减少数据库抵御操作系统崩溃或电源故障等损害的能力。1 级等同于异步（无 syncs()发生，存在较高的损害风险），等同于设置编译指示 synchronous=OFF。2 级是默认级别，存在较低的损害风险，等同于设置编译指示 synchronous=NORMAL。3 级降低了损害可能性，风险接近为零，但降低了写性能，等同于设置编译指示 synchronous=FULL。
- sqlite3BtreeSetPageSize：设置数据库页大小。
- sqlite3BtreeGetPageSize：返回数据库页大小。
- sqlite3BtreeSetAutoVacuum：设置数据库自动清理空闲页属性。
- sqlite3BtreeGetAutoVacuum：获取数据库是否自动清理页。
- sqlite3BtreeSetBusyHandler：设置繁忙句柄。

还有很多函数在 btree.h 和 btree.c 文件中有很详细的说明。这里列出的只是 B-tree 层实现的一部分 API，介绍这些 API，是为了帮助理解以及了解 B-tree 层可以做些什么。

311 显示类型、存储类以及亲缘性介绍

第 4 章讨论了 SQLite 存储类，并讨论了基本数据类型（或域）如何灵活地实现其他系统或数据库中数据类型的概念，进一步探索基本存储类机制非常有用，包括 SQLite 的显示类型和类型亲缘性的实现。

显示类型

SQLite 支持显示类型。稍作研究就会发现，显示类型存在多种解释。在编程语言中，显示类型就是如何定义或决定一个变量或值。下面有两种主要的解释：

- 显示类型是指变量的类型必须在代码中显式地声明。诸如 C/C++、Pascal 以及 Java 等语言都采用了显式类型定义。对于 Perl、Python 以及 Ruby 等动态类型定义语言，由于无须进行变量类型声明，所以没有显式类型定义。
- 显示类型是指变量完全无类型，而只有值有类型。这个描述基本上与动态定义语言相一致。一般而言，一个变量可以维护任何值，并且变量的类型在任何时候都是由当时它所维护的值决定的。因此，如果设 x='JujyFruit'，则这时 x 就是 text 类型。换句话说，如果 x 值符合整数定义并进行了整数的操作，则 x 就是整数类型。

为简便起见，将第一种解释记做 MT1，第二种解释记做 MT2。刚开始，可能不太容易看出哪个解释最适合 SQLite。例如，考虑下表：

```
create table foo( x  integer,
                  y  text,
                  z  real );
```

假设此时向该表插入如下一条记录：

```
insert into foo values ('1', '1', '1');
```

当 SQLite 创建该记录时，x、y 和 z 在存储内部会采用什么类型？答案是 integer、text 以及 real。如此看来，SQLite 好像采用 MT1：变量的类型已定义。但在 SQLite 中，列类型在定义中是可选项，可以在定义时不指定，因此，可以简单地定义 foo 表如下：

```
create table foo(x, y, z);
```

现在，做相同的插入记录操作：

```
insert into foo values ('1', '1', '1');
```

其中，x、y 和 z 的是什么类型呢？答案是：text、text 和 text，这是 SQLite 列的默认类型。以此假设，考虑下面对同一个表进行的插入处理语句：

312

```
INSERT INTO FOO VALUES (1, 1.0, x'10');
```

在这一行里，x、y 和 z 又是什么类型呢？integer、real 以及 blob。这符合 MT2：值本身决定了类型。

那么 SQLite 究竟采用了哪一个解释？简单回答都不是，也都是。详细的回答更复杂。关于 MT1，SQLite 支持声明列类型，这也是其他数据库的做法。但 SQLite 也可不声明列类型，这同 MT1 解释冲突。由于 SQLite 能在所有的情况下接受任何值并以此推断值类型，所以就无须声明列类型以帮助理解值类型。关于 MT2，SQLite 允许值类型“影响”（可能未完全确定）列存储的类型。但也可以声明列类型，该类型将对列值类型产生影响，所以也同 MT2 解释相冲突：类型仅由值决定。这里真正采用的是 MT3——SQLite 解释，它兼用了 MT1 和 MT2。

有趣的是，显示类型并没有阐述关于类型的全部问题，它仅针对类型声明和类型解析。类型的检查是如何进行的？如果声明一个列为 integer 类型，究竟是什么意思？

首先，考虑大多数其他关系数据库的做法。它们进行严格的类型检查，以此作为标准的域完整性的一部分，并且只有同类型的值才能存进该域。最后，也可使用额外的域限制，但无论如何不可以插入其他类型的值。考虑下述 Oracle 的示例：

```
SQL> create table domain(x int, y varchar(2));
Table created.

SQL> insert into domain values ('pi', 3.14);
insert into domain values ('pi', 3.14)
                           *
ERROR at line 1:
ORA-01722: invalid number
```

“pi”值并非整数类型值，而 x 列被声明为 int 类型。由于 x 发生完整性冲突导致整个插入操作被终止，所以连 y 列的错误都不会被提示。而在 SQLite 上执行此插入操作，将是声明一套，执行一套（声明与检查各自为政），并且 SQLite 不会终止处理：

```
sqlite> create table domain (x int, y varchar(2));
sqlite> INSERT INTO DOMAIN VALUES ('pi', 3.14);
sqlite> select * from domain;
x    y
---- -----
pi   3.14
```

SQLite 域完整性不包括严格的类型检查。那么，为什么这么做？列类型声明还有意义吗？当然有意义。那又是如何使用列类型声明的呢？将在类型亲缘性中使用。

313 简言之，SQLite 的显示类型表明，列可以有类型且类型可依据值来推断。类型亲缘

性阐述了这两者间的关系。类型亲缘性极具平衡艺术，它介于严格类型和动态类型之间。

类型亲缘性

在 SQLite 中，列没有类型或域的概念。虽然可以定义列类型，但在内部实现中采用了类型亲缘性。类型的声明和类型的亲缘性是不同的，类型亲缘性决定了 SQLite 存储列值的存储类。给定列值的实际列存储类是随值存储类和列亲缘性而变的。在研究该变化是如何工作之前，首先探讨列如何获取亲缘性。

列的类型和亲缘性

首先，每个列都有亲缘性，可分四类：数值、整数、文本以及空。列亲缘性直接由声明的类型（可空缺）决定。因此，当表中声明一个列时，所选择的列类型将最终决定列亲缘性。SQLite 依据下列规则设置列亲缘性：

- 列默认亲缘性类型是数值。如果一个列不是整数、文本或空，则将自动赋予该列的亲缘性为数值。
- 如果列声明类型含“int”（不分大小写），则该列的亲缘性为整数。
- 如果列声明类型含“char”、“clob”或“text”（不分大小写），则该列的亲缘性为文本。值得注意的是，“varchar”包括“char”，因此，也将赋予文本亲缘性。
- 如果列声明类型含“blob”（不分大小写），或没有声明类型，则将赋予该列亲缘性为空。

■**注意**：关注默认处理，如，浮点数亲缘性为整数。如果没有声明列类型，亲缘性将为空，这时，所有的值将以给定的存储类存储（或者依据数据判断）；如果不能确定存放在列中的数据类型，或放弃声明，而在打开数据时改变亲缘性类型，空将是最好的亲缘性。然而，应该注意声明为空、文本或整数都不匹配的类型。这种情况下，虽然可能凭直觉认为亲缘性为空，但实际亲缘性却是数值。例如，如果定义了 JUJYFRUIT 类型的列，则亲缘性不会是空，因为 SQLite 不能识别该类型，相反，亲缘性将为数值（有趣的是，当声明一个列类型为数值时，亲缘性也会是数值）。除非使用一个以数值结尾的无法识别的类型，这可能会完全放弃列类型声明，并确保声明列亲缘性为空。

314 亲缘性和存储

每个亲缘性影响了值在关联列中如何存储。存储管理规则如下：

- 数值列可有 5 种存储类。数值列偏向于数值存储类（整数和实数）。当一个文本值插入数值列时，会试图将其转成整数存储类。假如失败，将试图转成实数存储类。再次失败，将采用文本存储类存储值。
- 整数列将尽可能参照数值列。整数列实际将存储实数值，但如果实数值没有小数部分，将采用整数存储类存储。整数列也会试图存储文本值，并试图以整数形式存储，否则，文本值将以文本类型存储。
- 文本列将所有的整数值和实数值转换为文本类型。
- 空列不会试图转成任何值。所有的值都采用指定存储类存储。
- 列不会试图转换空值或 Blob 值——不管亲缘性如何。空和 Blob 值始终存储于每列中。

这些规则初看起来可能会比较复杂，但是整体设计目标很简单：如果需要，可使 SQLite 模仿其他关系数据库。也就是说，如果将 SQLite 的列当成传统的数据库列，类型亲缘性规则按照传统方式处理。如果声明了整数类型列，并存放了整数，则将以整数存储。如果声明了文本、字符或可变长字符类型列，并存放了整数，则建议文本存储。不过，不必受限于这些约定，SQLite 会以合理的方式存储。

执行中的亲缘性

参看以下例子，以熟悉亲缘性如何工作。考虑以下示例：

```
sqlite> create table domain (i int, n numeric, t text, b blob);
sqlite> insert into domain values (3.142, 3.142, 3.142, 3.142);
sqlite> insert into domain values ('3.142', '3.142', '3.142', '3.142');
sqlite> insert into domain values (3142, 3142, 3142, 3142);
sqlite> insert into domain values (x'3142', x'3142', x'3142', x'3142');
sqlite> insert into domain values (null, null, null, null);
sqlite> select rowid, typeof(i), typeof(n), typeof(t), typeof(b) from domain;

rowid       typeof(i)   typeof(n)   typeof(t)   typeof(b)
----------  ----------  ----------  ----------  ----------
1           real        real        text        real
2           real        real        text        text
```

```
3           integer     integer     text        integer
4           blob        blob        blob        blob
5           null        null        null        null
```

第一个插入语句插入了实数类型值，这是由插入语句的格式和查询语句返回的typeof(b)列的显示结果类型决定的。请记住，blob 列采用空存储类型，不会尝试转换输入数据的存储类，因此，b 列采用了与插入语句中定义相同的存储类。i 列保持数值存储类，即只要允许，就转成数值类型。n 列不需转换成其他存储类型。t 列转成了文本类型。b 列同存储上下文中给定的完全一致。之后的每次插入操作显示了转换规则在不同情况下的应用。下面的 SQL 说明了存储类排序顺序以及组间比较（这由同一套规则管理）：

315

```
sqlite> select rowid, b, typeof(b) from domain order by b;
rowed    b         typeof(b)
-----    ------    ---------
5        NULL      null
1        3.142     real
3        3142      integer
2        3.142     text
4        1B        blob
```

由此可见，空值将排在第一个，其次是整数和实数，接着是文本，然后是 blob 类型。下面的 SQL 表明这些值是如何和整数 1000 进行比较的。b 中的整数和实数小于 1000，因为它们是数值比较，但是文本及 blob 比 1000 大，因为它们使用更高的存储类。

```
sqlite> select rowid, b, typeof(b), b<1000 from domain order by b;

rowed    b         typeof(b)         b<1000
-----    -----     --------      ----------
5        NULL      null          NULL
1        3.142     real          1
3        3142      integer       1
2        3.142     text          0
4        1B        blob          0
```

类型亲缘性和严格类型间的主要区别在于，类型亲缘性从不对不兼容类型进行冲突约束。SQLite 将为所有列中存入的任何值寻找一个数据类型。唯一值得考虑的是，应该使用什么样的类型。SQLite 中列声明类型的唯一角色就是简单地决定亲缘性，最终，列的亲缘性能够做到如何存储值。当然，SQLite 确实支持确保列只接受一个给定类型或一

个类型范围。通过检查约束，解释本段后面侧栏的“临时类型限制”。

存储类及类型转换

关于存储类，另一个需要注意的是，存储类有时也会影响值的比较方式。具体地说，比较前，SQLite 有时将在数值存储类（整数和实数）和文本存储类之间进行值转换。对于二进制比较，采用如下规则。

316

规则 1：当一个列值与表达式结果比较时，列的亲缘性在比较前应用于表达式结果。

规则 2：当两个列值比较时，如果一个是整数或数值亲缘性列，而另一个不是，则数值亲缘性将用于非数值列的文本值。

规则 3：当两个表达式比较时，SQLite 不会采取任何转换，而是对结果进行比较。如果表达式类似于存储类，则用与该存储类关联的比较函数进行值的比较。否则，采用它们的基本存储类进行比较。

注意，此处的术语“表达式”指的是任何标量表达式或文字表达式，而不是列的值。为了阐述规则 1，考虑以下几点：

```
sqlite> select rowid,b,typeof(i),i>'2.9' from domain order by b;
rowed    b        typeof(i)    i>'2.9'
-----    -----    --------     ------------
5        NULL     null         NULL
1        3.142    real         1
3        3142     integer      1
2        3.142    real         1
4        1B       blob         1
```

表达式 '2.9' 虽是文字，但在比较前被转成了整数。所以，列根据值是什么进行解释。如果“2”是非数值字符会如何？然后 SQLite 会返回比较存储类，其中整数和数值类型总是小于文本：

```
sqlite> select rowid,b,typeof(i),i>'text' from domain order by b;

rowid    b        typeof(i     i>'text'
-----    -----    --------     ------------
5        NULL     null         NULL
1        3.14     real         0
3        314      integer      0
2        3.14     real         0
```

```
4        1B       blob          1
```

规则 2 简单阐述了比较一个整数和非整数列时，SQLite 会尝试将非整数列转换成整数形式。

```
sqlite> create table rule2(a int, b text);
sqlite> insert into rule2 values(2,'1');
sqlite> insert into rule2 values(2,'text');
sqlite> select a, typeof(a),b,typeof(b), a>b from rule2;

a           typeof(a)    b           typeof(b)    a>b
----------  ----------   ----------  ----------   ----------
2           integer      1           text         1
2           integer      text        text         0
```

列 a 是一个整数，b 是文本。当计算表达式 a>b 时，SQLite 尝试强制将 b 转成整数。317在第 1 行中，b 是“1”，该值可被强制转成整数。SQLite 负责转换并比较整数。在第 2 行中，b 是“text”，不可以被转换。SQLite 接着比较整数和文本存储类。

规则 3 仅仅重申由上下文创建的存储类只比较表面值。如果一个文本类型值同一个整数类型值比较，则文本值更大。

此外，可以通过 cast()函数手动转换一个列或一个表达式的存储类型。考虑如下例子：

```
sqlite> select typeof(3.14), typeof(cast(3.14 as text));

typeof(3.14)   typeof(cast(3.14 as text))
------------   --------------------------
real           text
```

临时类型限制

如果为了域完整性，需要有比类型亲缘性更强的手段，那么可通过 CHECK 约束提供支持，可直接通过单个内建函数和一个 CHECK 约束来实现伪类型限制。如前所述，SQLite 的 typeof()函数可返回值的存储类，可在任何关系表达式中使用 typeof()来测试一个值的类型。下面是一个例子：

```
sqlite> select typeof(3.14) = 'text';
0
sqlite> select typeof(3.14) = 'integer';
0
```

```
sqlite> select typeof(3.14) = 'real';
1
sqlite> select typeof(3) = 'integer';
1
sqlite> select typeof('3') = 'text';
1
```

因此，可利用该函数实现 CHECK 约束，以便限制一个列中可接受的类型：

```
sqlite> create table domain (x integer check(typeof(x)='integer'));
sqlite> insert into domain values('1');
SQL error: constraint failed

sqlite> insert into domain values(1.1);
SQL error: constraint failed

sqlite> insert into domain values(1);
sqlite> select x, typeof(x) from domain;
```



```
x   typeof(x)
--  ----------
1   integer
sqlite> update domain set x=1.1;
SQL error: constraint failed
```

这种方式的唯一缺点是，只能检查 SQLite 的原始存储类（或者，可通过其他内建 SQL 函数实现）。不过，如果是程序员，并使用支持 SQLite 用户定义函数的可扩展语言（如，Perl、Python 或 Ruby）或者直接使用 SQLite 的 C API，可以实现类型检测更详尽的函数，该函数可由 CHECK 约束内部调用。

预写日志

随着 SQLite 3.7.0 的发布，引入了新的可选管理原子事务的模式：预写日志。预写日志（Write Ahead Log，WAL）对于广泛的数据库技术界来说，并不新鲜，但是，包括在 SQLite 中有两个值得惊叹的原因：首先，进一步增强 SQLite 和数据库的精密性与鲁棒性；其次，如此巨大的特性已加入到 SQLite 后，代码和二进制文件依然如此紧凑和简洁。

WAL 工作原理

传统的操作模式中，SQLite 使用回滚日志，从 SQL 状态集中捕获预修改数据，然后修改数据库文件（这里避开讨论变量缓存以及文件系统层，是因为它们和模型无关）。使用 WAL 时，将反过来处理。取代写入原始信息模式，预修改数据放入回滚日志，利用 WAL 放弃原始未关联数据库文件的数据。WAL 文件用来记录给定事务导致的数据变化。一个提交操作变为特殊的写进 WAL 的记录，以表明前面的修改确实完成，并实现了 ACID 原则。

数据库文件和日志文件之间的角色变化会立刻改变事务的动态性能。不用在数据库文件中竞争相同的数据库页，在 WAL 中，多事务可以同时记录它们数据的改变，并能够持续从数据库中读取未改变的数据。

检查点

对于第一个问题，很多人面对 WAL 技术时通常思考“可是，什么时候数据最终写入数据库？”这个问题很好。显然，一个正在写入不断增长的 WAL 文件的没有结束的数据变化流，无法能扩展或承担难以预测的文件系统故障。WAL 使用检查点函数将修改写回数据库。这个处理自动进行，所以开发者无须关注自己如何管理检查点和数据库写回处理。默认情况下，当 WAL 文件发现有 1000 页修改时，将调用检查点。可修改检查点执行标准的参数以适应不同的操作场合。

并发

319

因为现在的修改已写入 WAL 文件，而不是数据库文件，您可能想知道这些是如何影响并发的。好的消息是：“绝对比以前更好！”

从启用了 WAL 的数据库中进行数据读取的操作会在 WAL 文件中开始查找最后一条记录。这点成为只读一致性的结束标志，这样，就不会考虑新的写数据操作超越文件的该点。这也是读操作实现一致性读的原因——参考结束标记，不用考虑提交后的变化。每个读操作的访问都是独立的，所以可能有多个线程，每个线程在 WAL 文件中都有属于自己的独立结束标志的符号。

访问一个数据页，读操作采用 WAL 索引结构来扫描 WAL 文件，以确认修改页是否存在，如果能找到，则采用最近的版本。如果修改页在 WAL 文件中不存在，意味着从上个检查点至今未发生修改，并且读操作是从数据库文件访问数据页的。

WAL 索引结构在共享内存中实现，意味着所有的线程和进程必须访问相同的内存空间。也就是说，程序必须都在同一台机器上执行，目的是访问启用 WAL 的数据库，或更准确地说，是使用 WAL 索引。这也是操作无法访问像 NFS 类的网络文件系统的原因。

至于操作，受益于 WAL 文件序列化属性，可以简单地将修改添加到文件末端。但要注意，这意味着必须轮流添加修改，所以写操作是阻塞型写。

激活和配置 WAL

SQL 依然会默认采用回滚日志，除非您决定使用 WAL 数据库。同其他编译指示 PROGMA 不同，启用 WAL 是数据库级持久性改变。这意味着，可以在程序里启用 WAL，或从 SQLite 命令 shell 启用，并且之后其他程序将在 WAL 模式下使用数据库。这点非常方便，因为这意味着无须改变应用程序就可启用 WAL。

设置 PRAGMA journal_mode=WAL 是激活 WAL 的必要命令。该编译指示的调用将以字符串形式返回数据库最终的日志模式。

```
SQLite version 3.7.3
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> PRAGMA journal_mode=WAL;
wal
```

如果 WAL 改变成功，将返回字符串 wal，如上所示。如果由于某种原因失败了，诸如底层主机不支持必要的共享内存要求，日志模式将不会改变。例如，在默认的数据库中，这意味着命令将返回字符串 delete，即回滚日志依然有效。

320 与 journal_mode 编译指示相同，有几个 API 选项和相关的编译指示也支持控制 WAL 和检查点行为。

- sqlite3_wal_checkpoint()：强制设立检查点（也可作为 wal_checkpoint 编译指示用）。
- sqlite3_wal_autocheckpoint()：改变 autocheckpoint 页阈值（wal_autocheckpoint 编译指示）。
- sqlite3_wal_hook()：注册一个回调函数，当程序提交至 WAL 时调用。

通过这些选项可以对 WAL 和检查点行为进行微调，以满足各种应用，设置包括最

不常见的应用。

WAL 的优缺点

很显然，WAL 的变化带来优点的同时也会带来缺点。优点是重要的，但最好是知道这些优点，以便形成自己的判断力。

WAL 有如下优点：

- 读操作不会阻塞写操作，同时写操作也不会阻塞读操作。这是并发管理的“黄金准则”。
- 在大多数操作场景中，与回滚日志相比，WAL 相当快。
- 磁盘 I/O 变得更可预见，并且引起更少的 fsync()系统调用。因为所有的 WAL 写操作是线性写入日志文件，很多 I/O 变得连续并能够按计划执行。

下面是 WAL 值得关注的一些缺点：

- 所有的处理被绑定到单个主机上。也就是说，不能在如 NFS 这样的网络文件系统上使用 WAL。
- 为满足 WAL 和相关共享内存的需要，使用 WAL 引入了两个额外的半持久性文件<yourdb>-wal 和<yourdb>-shm。这对于那些使用 SQLite 数据库作为应用程序文件格式是不具有吸引力的。这也影响了只读环境，因为-shm 文件必须是可写的，并且/或数据库所在目录也必须是可写的。
- 对于非常大（接近 GB 级）的事务，WAL 的性能将会降低。虽然 WAL 是一个高性能选项，但是非常大或运行时间非常长的事务会引入额外的开销。

如果考虑在不寻常的环境中使用启用了 WAL 的 SQLite 数据库，大量的其他边界情况需要研究。您可在 www.sqlite.org/wal.html 上查看更多关于这些问题的细节。

启用 WAL 时 SQLite 数据库的操作问题

321

启用 WAL 的 SQLite 数据库工作时，需要考虑一些操作，这些操作主要是有关性能和恢复的问题。

性能

前面我们注意到，启用 WAL 通常会获得更好的性能。对于数据非常大的改动或者

运行时间较长的事务，可能导致性能轻微下降，这是缺点。这是如何发生以及要做什么的背景。

启用 WAL，隐含获得比回滚日志好的性能，因为提交事务只需一个写操作，而不是两个（一个数据库和一个回滚日志）。然而，直到一个 WAL 文件中释放空间的检查点产生，文件自身必须持续增长，因为要记录越来越多的改动。这通常不是问题：有人会是幸运的提交者，将检查点阈值设置为 1000 页来推送 WAL，并且 WAL 文件将清空，如此重复。但是当检查点发现任何当前活动读操作的终结点标识时，它将完成其工作。这意味着，运行时间较长的事务或者读活动能够防止检查点前进，甚至在随后的尝试中。反过来，意味着 WAL 文件大小会持续增长，同时新的读操作寻找页的时间也会相应地变长。

另一个性能考虑是，当事务达到检查点阈值时，有多少工作被触发。你可能会发现在某些情况下，许多非常快的事务的变量性质，一个为后继者必须执行检查点工作的稍长事务，变得不引人注意了。

对于这两个性能假设，从根本上估算读写性能。推荐的方法是：首先不可能两者兼得，因此，需要用折中方案，如果需要，可使用 sqlite3_wal_checkpoint()和 sqlite3_wal_autocheckpoint()工具平衡负载，以找到好的平衡。

恢复

SQLite 3.7.0 之前的版本没有 WAL 概念，也没有基于 WAL 的数据库恢复方法。为了防止之前版本在崩溃恢复过程中干扰启用了 WAL 的 SQLite 数据库，数据库文件格式号从 1 跳到 2。当一个旧版本的 SQLite 试图使用这一变化恢复数据库时，它会意识到这不是一个“有效”的 SQLite 数据库，并且会报告类似如下的错误：

```
file is encrypted or is not a database
```

此时无须惊慌，数据库是完好的，但是 SQLite 的版本需要升级才能处理该数据库，或者可以回到回滚日志。

```
PRAGMA journal_mode=DELETE;
```

这样将数据库文件格式号重置为 1。

总结

到这里，本章与全书的 SQLite 旅程就结束了，希望您能够喜欢 SQLite。正如在第 1 章中看到的，SQLite 不仅仅是一个免费数据库，它还是一款编写得很好的软件库，有着广泛的应用。它是一个数据库和一个实用工具，也是一个有用的编程工具。

本章所看到的只是内部机制的皮毛，但它们使你对 SQLite 是如何工作的有了更好的了解。也让您了解了 SQLite 是如何优雅地解决非常复杂的问题的。您知道 SQL 有多强大，同时，也看到它背后的模型有多复杂。但是，SQLite 是一个小型库，它将很多概念放在少量代码中工作，并且表现优异！

索 引

特殊字符

$argc parameter（参数$argc），225
$func_ref parameter（参数$func_ref），225
$name parameter（参数$name），225
$name variable（变量$name），250
$type_id variable（变量$type_id），250
$x variable（变量$x），245
% operator（%操作符），61
& operator（&操作符），61
* operator（*操作符），61
rowid value（_rowid_值），94
| operator（|操作符），61
|| operator（||操作符），61
+ operator（+操作符），61
< operator（<操作符），61
<< operator（<<操作符），61
<= operator（小于操作符），61
<> operator（不等于操作符），61
!= operator（不等于操作符），61
= operator（等于操作符），61
== operator（相等操作符），61
> operator（大于操作符），61
>= operator（大于等于操作符），61
>> operator（>>操作符），61
/ operator（/操作符），61
- operator（-操作符），61

A

abort policy（中止策略），113
abort resolution（中止解决方法），113 - 115
abs() function（绝对值函数），66
access functions, for B-tree module API（B-tree 模块 API 的访问函数），308 - 309
activation, of WAL（预写日志的激活），319
add column clause（添加字段子句），55
ADD COLUMN command（添加字段命令），12
ADD CONSTRAINT command（添加约束命令），12
Add File dialog box（添加文件对话框），263
Additional Dependencies property page（额外依赖属性页），28
administration（管理），118 - 124
 attaching databases（附着数据库），118 - 119
 cleaning databases（清理数据库），119
 of databases（数据库相关），35 - 45
 backing up database（备份数据库），42 - 43
 creating database（创建数据库），35 - 37
 database file information(数据库文件信息），44 - 45
 exporting data（导出数据），39
 formatting data（格式化数据），40 - 41
 getting schema information(获取模式信息），37 - 39

importing data（导入数据），40
unattended maintenance（无人值守维护），41 - 42
explain query plan command（查询计划解释命令），123
and pragmas（编译指示），120 - 122
auto_vacuum（自动清理），122
cache size（缓存大小），120
database schema（数据库模式），120 - 121
integrity_check（完整性检查），122
synchronous（同步），121 - 122
temp_store（临时存储），122
temp_store_directory（临时存储目录），122
sqlite_master table（sqlite_master 表），123
administrators, SQLite for（管理员的 SQLite），3
Advanced tab（高级标签页），19
after keyword（after 关键字），108
after trigger（after 触发器），110
after value（after 值），134
aggregates，67
in Extension C API（扩展 C API 中的聚合），204 - 209
example of（聚合例子），206 - 209
Registering（注册），205 - 206
with language extensions（语言扩展）
for Perl（Perl），225 - 226
for Python（Python），231 - 232
user-defined（用户自定义），137
aliases, for tables（表的别名），77 - 79
ALTER COLUMN command（修改列命令），12
ALTER TABLE command（修改表命令），12，54
alter table statement（修改表语句），286，288
altering tables（修改表），54 - 55
ambiguity, of column names（列名称的歧义性），77
aName variable（变量 aName），272
AND operator（与操作符），61
Android DDMS（Android DDMS），280
Android Developer Tools（Android 开发者工具），280 - 281
Android development（Android 开发），279 - 301
backup service for databases（数据库备份服务），300
and large databases（大数据库），300 - 301
prerequisites for（必备条件），279 - 284
additional components（额外组件），281 - 284
downloading Android DeveloperTools（下载 Android 开发者工具），280 - 281
downloading Android SDK Starter Package（下载 Android SDK Starter 包），280
JDK（Java 开发套件），280
Seinfeld Android app（Seinfeld Android 应用），294 - 300
adding database to project（将数据库添加到工程），296
creating project（创建工程），295 - 296
linking data and user interface（连接数据与用户接口），298 - 299
querying foods table（查询 foods 表），296 - 297
user interface for（用户接口），297 - 298
viewing app（查看应用），299 - 300
SQLiteDatabase class（SQLiteDatabase 类），286 - 290
convenience methods for（便捷方法），288 - 289
executing queries with（执行查询），287 - 288
implementing（实现），290 - 292
opening and closing databases With（打开与关闭数据库），286 - 287
transactions with（事务），289
useful methods for（有用的方法），290
SQLiteOpenHelper class（SQLiteOpenHelper 类），285 - 286，290 - 292

SQLiteQueryBuilder class（SQLiteQueryBuilder 类），293－294
Android Platforms component（Android 平台组件），282
Android Virtual Device (AVD)（Android 虚拟设备），283
android.database.sqlite namespace（命名空间 android.database.sqlite），285
API, for B-tree module（B-tree 模块 API）
access functions（访问函数），308－309
configuration functions（配置函数），310
cursor functions（游标函数），309
record functions（记录函数），310
table functions（表函数），309
Appcelerator Titanium framework（Appcelerator Titanium 框架），258，284
Application Settings（应用程序设置），26，28
applicationDidFinishLaunching event（applicationDidFinishLaunching 事件），276
applicationDidFinishLaunching Function（applicationDidFinishLaunching 函数），269
applicationWillTerminate event（applicationWillTerminate 事件），272
apt utility（apt 程序），31
apt-get utility（apt-get 程序），31
architecture（架构），5－8
back end（后端），7－8
compiler（编译器），6
interface（接口），6
utilities（实用程序），8
virtual machine（虚拟机），6－7
Arithmetic operators（算术操作符），62
as keyword（关键字 as），79
asc sort order（升序排序），65
atomic principle（原子原理），111
attach command（附着命令），118
ATTACH event（ATTACH 事件），182
attach statement（附着语句），184
attaching databases（附着数据库），118－119
auto_vacuum pragma（编译指示自动清理），119，122，128
autocommit mode, and write transactions（自动提交模式，写操作事务），143－145
autoincrement column（自增长字段），95
autoincrement constraint（自增长约束），95－96
autoincrement keyword（关键字 autoincrement），94－95
available_drivers() function（available_drivers() 函数），222
AVD (Android Virtual Device)（AVD（Android 虚拟设备）），283
avg() function（avg()函数），67，137，195
avg aggregate（avg 聚合），67

B

b column（b 列），317
back end，architecture of SQLite（SQLite 的后端，架构），7－8
backing up（备份），42－43
BackupAgentHelper API（BackupAgentHelper API），300
BackupManager API（BackupManager API），300
Bakery row（面包店行），64
bar.db database（数据库 bar.db），10－11
base table（基本表），54
before keyword（关键字 before），108
before trigger（before 触发器），110
begin command（开始命令），111，116－118，139，141
begin exclusive command（开始独占事务命令），118，148
begin immediate command（立即开始事务命令），118，148－149
begin line（开始行），141

begin statement（开始语句），117
begin.commit command（开始提交命令），141
begin...commit/rollback transaction scope（开始...提交/回滚事务范围），112
beginTransaction() method（beginTransaction()方法），249，289
beginTransactionWithListener() method,（beginTransactionWithListener()方法），289
binaries（二进制包）
 for Linux（Linux 系统上的），30 - 31
 for Mac OS X（ Mac OS X 系统上的），30 - 31
 for POSIX systems（POSIX 系统上的），30 - 31
binary collation（二进制排序规则），101，138，216
Binary large object (BLOB)（二进制大对象），4，102
binary operators(二进制操作符)，for where clause（where 子句），60 - 62
bind_param() method（bind_param() method 方法），234
bind_params() method（bind_params() 方法），234
BLOB (Binary large object)（二进制大对象），4，102
blob class（blob 类），102 - 103，196
blob columns（blob 列），315
B-tree module（B-tree 模块），303 - 310
 API for（API）
 access functions（访问函数），308 - 309
 configuration functions（配置函数），310
 cursor functions（游标函数），309
 record functions（记录函数），310
 table functions（表函数），309
 data structures of SQLite（SQLite 数据结构），127
 and database file format（数据库文件格式）
 B+trees(B+tree)，304 - 305
 field types in（字段类型），305 - 306
 hierarchical data organization In(层次化数据组织），307
 overflow pages in（溢出页），307 - 308
 page reuse in（页面重用），304
 record structure in（记录结构），305
 records in,（记录），304
btree.c file（btree.c 文件），310
btree.h file（btree.h 文件），310
Build and Run menu option（构建与运行菜单项），272
buildUnionQuery() method（buildUnionQuery()方法），294
busy() function（busy()函数），147
busy conditions（繁忙情况），error handling in C API of（C API 中的错误处理），176 - 177
busy handler，using with transactions（事务相关的繁忙处理），146 - 147

C

C API，153 - 193
 error handling in（错误处理），174 - 176
 of busy conditions（繁忙情况），176 - 177
 of schema changes（模式改变），177 - 178
 fetching records with（获取记录），164 - 169
 column information for（列信息），165 - 166
 column values in（列值），166 - 167
 example of（示例），168 - 169
 operational control functions（操作控制函数），178 - 189
 commit_hook() function（commit_hook()函数），178 - 179
 rollback_hook() function（rollback_hook()函数），179

set_authorizer() function（set_authorizer()函数），180 - 189
update_hook() function（update_hook()函数），179 - 180
parameterized queries with（参数化查询），169 - 174
named parameters（命名参数），173
numbered parameters（编号参数），172
Tcl parameters（Tcl 参数），173 - 174
prepared queries with（准备查询），161 - 164
compilation of（编译），161 - 162
execution of（执行），162
finalization of（结束），163 - 164
threading support in（线程支持），190 - 193
and memory management（内存管理），193
shared cache mode（共享缓存模式），190 - 193
wrapper functions in（封装函数），153 - 160
for connecting and disconnecting（连接与断开），153 - 155
exec() function（exec()函数），155 - 159
get_table() function（get_table()函数），159 - 160
cache partition size parameter（缓存分区大小参数），284
cache_size pragma（编译指示缓存大小），120，145
Cartesian join（笛卡儿连接），75
cascade action（级联行为），101
case expression（case 表达式），83 - 84
cast() function（cast()函数），212，317
cast table（cast 表），115
cast.name column（cast.name 列），115
check constraints（check 约束），99 - 100，137，315，317 - 318
Checkpoints（检查点），and WAL（预写日志），318
Churcher（教堂），Claire（克莱儿），47
class values（类值），103
classes folder（Classes 文件夹），265
-classpath option（-classpath 项），237
clauses（子句）
distinct clause（distinct 子句），72
group by clause（group by 子句），67 - 71
limit clause（limit 子句），64
order by clause（order by 子句），64 - 66
where clause（where 子句），59 - 64
binary operators（二进制操作符），60 - 62
glob operator（glob 操作符），64
like operator（like 操作符），63
logical operators（逻辑操作符），63
operators（操作符），60
values（值），60
cleaning databases（清理数据库），119
cleanup handlers（清理处理），for functions in Extension C API（扩展 C API 中的函数），202 - 203
clone command（克隆命令），23
close() method（close()方法），228，244，285，287
close instruction（关闭指令），151
closing databases()，with SQLiteDatabase class（用 SQLiteDatabase 类关闭数据库），286 - 287
CLP (command-line program)（CLP 命令行程序），17，32 - 35
in command-line mode（在命令行模式中），34 - 35
installing on Windows（在 Windows 上安装），18 - 21
in shell mode（在 shell 模式中），33 - 34
cnx argument（参数 cnx），197
coalesce function（coalesce 函数），85 - 86
code and SQLite（代码与 SQLite），149 - 152

importance of finalizing（结束的重要性），150 - 151
shared cache mode（共享缓存模式），151 - 152
using multiple connections（使用多个连接），149 - 150
collate keyword（关键字 collate），101
collate nocase constraint（collate nocase 约束），54
collate1.sql file（collate1.sql 文件），209
collate2.sql file（collate2.sql 文件），210
collation interface（排序规则接口），199
collation_name parameter（参数 collation_name），119
collations（排序规则）
for columns, and indexes（列与索引），107
and domain integrity（域完整性），101
in Extension C API（扩展 C API 中），209 - 217
defined（定义），210 - 212
on demand（随需而应），216 - 217
example of（示例），212 - 216
overview of（概要），210 - 212
types of（类型），212
user-defined（用户自定义），138
column constraints（列约束），54
column names（类名称），and ambiguity of（歧义性），77
column types（列类型），and type affinity（类型亲缘性），313
column_def attribute（属性 column_def），55
column_definitions attribute（属性 column_definitions），54
column_list variable（变量 column_list），87
columns（列）
collations for（排序规则），and indexes（索引），107
information about（有关信息），in C API（C API 中），165 - 166
values of（值），in C API（C API 中），166 - 167
columns() method（columns()方法），239
columns variable（变量 columns），106
command-line program. *See* CLP（命令行程序，参考 CLP）
commands（命令）
delete command（删除命令），92
explain query plan command（查询计划解释命令），123
insert command（插入命令），87 - 91
multiple rows（多行），90 - 91
one row（一行），87 - 89
set of rows（一组），89 - 90
select command（选择命令），57 - 59
syntax for SQL in SQLite（SQLite 中的 SQL 语法），51 - 52
update command（更新命令），91 - 92
comma-separated values (CSV)（逗号分隔的值），40
comments, syntax for SQL in SQLite（SQLite 中的 SQL 语法，注释），53
commit() method（commit()方法），249
commit command（提交命令），111，115，118，139，141，143
commit line（提交行），141
commit_hook() function，in C API（C API 中 commit_hook()函数），178 - 179
compactness, features of SQLite（SQLite 的紧凑性），9
compilation, of prepared queries ()（准备查询的编译），161 - 162
compile() method（compile()方法），239
compiler（编译器），6
compileStatement() method（compileStatement()方法），288
compound queries（组合查询），81 - 83
con.commit() lines（con.commit()行），230
concurrency, and WAL（并发与预写日志），319

Configuration drop-down box（下拉配置框），26，28
configuration functions, for B-tree module API（B-tree 模块 API 中的配置函数），310
configure command（配置命令），30
conflict resolution, and transactions（冲突解决与事务），112－115
connect() function（connect()函数），227
connecting to SQLite, using language extension（使用扩展语言连接 SQLite）
 for Java（Java），238
 for Perl（Perl），222
 for PHP（PHP），248
 for Python（Python），227
 for Ruby（Ruby），233
 for Tcl（Tcl），244
Connection::create_aggregate() method（Connection::create_aggregate()方法），231
connections, data structures of SQLite（SQLite 的数据结构，连接），126－127
constraints（约束）
 of check（检查约束），99－100
 of foreign key（外键约束），100－101
 of not null（非空约束），98－99
 of primary key（主键约束），94－97
 of unique（唯一性约束），93
contacts database（数据库 contacts），93
contacts table（表 contacts），92，94，97
contacts.name column（列 contacts.name），101
Control Panel（控制面板），19
convenience, features of SQLite（SQLite 的便捷性），10－11
correlated subquery（相关子查询），80
:cosmo parameter（参数:cosmo），173
count() function（count()函数），67，69，137，195
count aggregate（count 聚合），67－68，72
count(*)<20 predicate（断言 count(*)<20），71
create index command（创建索引命令），115
Create Table As Select (CTAS)（从 select 命令创建表），91
create table command（创建表命令），53－55，88，90
create table statement（ 创建表命令），91，100，161，286,288
create_function() method（create_function()方法），230
CREATE_IF_NECESSARY behavior（如果有需要就创建行为），287
createaggregatefunction() function（createaggregatefunction()函数），137
crf() function（crf()函数），216
cross joins（交叉连接），75－76
cross product（交叉乘积），75
c:\sqlite directory（ c:\sqlite 目录），25，33
CSV (comma-separated values)（CSV，逗号分隔的值），40
CTAS (Create Table As Select)（CTAS，从 select 命令创建表），91
ctx argument（参数 ctx），198
current_date reserved word（保留字 current_date），98
current_time reserved word（保留字 current_time），98
current_timestamp reserved word（保留字 current_timestamp），98
cursor functions, for B-tree module API（ B-tree 模块 API 游标函数），309
Cursor.executemany() method（Cursor.executemany()方法），229

■D

data argument（参数 data），179，216
data definition language (DDL)（数据定义语言

DDL），38，53
data integrity（数据完整性），92 - 104
domain integrity（域完整性），97 - 101
check constraints（check 约束），99 - 100
and collations（排序规则），101
default values（默认值），98
foreign key constraints（外键约束），100 - 101
NOT NULL constraints（非空约束），98 - 99
entity integrity（实体完整性），93 - 97
primary key constraints（外键约束），94 - 97
unique constraints（唯一性约束），93
indexes（索引），106 - 108
and collations for columns（列的排序规则），107
utilization of（利用），107 - 108
storage classes for SQLite（SQLite 存储类），101 - 104
triggers（触发器），108 - 111
error handling with（错误处理），110
and updatable views（可更新的视图），110 - 111
update triggers（更新触发器），109 - 110
views in SQLite（SQLite 中的视图），104 - 106
Data Manipulation Language (DML)（数据库控制语言 DML），39，53，87
data source name (DSN)（数据源名称 DSN），248
data structures（数据结构），126 - 135
B-tree and pager（B-tree 与 pager），127
connections and statements（连接与语句），126 - 127
database administrator (DBA)（数据库管理员 DBA），3
database file format, and B-tree module（数据库文件格式，B-tree 模块）
B+trees（B+tree），304 - 305
field types in（字段类型），305 - 306
hierarchical data organization in（层次化数据组织），307
overflow pages in（溢出页），307 - 308
page reuse in（页面重用），304
record structure in（记录结构），305
records in,（记录），304
database schema pragma（编译指示 database schema），120 - 121
database_list pragma（编译指示 database_list），118，121
database_name parameter（参数 database_name），118 - 119
Database::create_function() method（create_function()方法），236
Database::get_first_row() method（Database::get_first_row()方法），235
Database::get_first_value() method（Database::get_first_value()方法），235
databaseName variable（变量 databaseName），271
databasePath variable（变量 databasePath），270
Database::prepare() method（Database::prepare()方法），233
Database::query() method（Database::query()方法），235
databases（数据库）
attaching（附着），118 - 119
backing up（备份），42 - 43
cleaning（清理），119
connecting to（连接），128
creating（创建），35 - 37，53 - 55
example（例子），47 - 50
installation（安装），48 - 49
running examples（运行示例），49 - 50
exporting data from（导出数据），39 - 41
file information for（文件信息），44 - 45
formatting data（格式化数据），40 - 41
importing data into（导入数据），40

querying（查询），55－86
daterepr property（daterepr 属性），241
db argument（参数 db），216
db parameter（参数 db），293
db2 database（数据库 db2），119
db2.foo table（表 db2.foo），119
DBA (database administrator)（数据库管理员），3
DBI::connect function（DBI::connect 函数），222
DBL (declarative burger language)（声明式汉堡语言），51
DDL (data definition language)（数据定义语言），38，53
DDMS, Android（DDMS, Android），280
deadlocks, and transactions（死锁与事务），116－117
declarative burger language (DBL)（声明式汉堡语言），51
declarative language（声明式语言），50
default constraint（默认约束），98－99
default keyword（关键字 default），98
default values（默认值），and domain integrity（域完整性），98
default_cache_size pragma(编译指示 缓存大小)，120
deferrable clause（可延迟的子句），101
deferred transaction type（延迟的事务类型），117
delete command（删除命令），92，108，111
delete operation（删除操作），135，179－180
Delete SQLITE_BUSY value（删除 SQLITE_BUSY 值），146
delete statement（删除语句），12，132，159，184
delete trigger（删除触发器），109，111
derived tables（继承表），80，104
desc sort order（降序排序），65
description value（描述值），83
detach command（detach 命令），119，185
DETACH event（DETACH 事件），182
detach statement（detach 语句），184
details view（详细视图），105
Developer Program, membership of（开发者计划，会员资格），254
Developer Tools, Android（Android 开发工具），280－281
developers（开发者），2－3
difference operation（差操作），82
disconnect method（disconnect 方法），222
distinct clause（distinct 子句），72
distinct keyword（关键字 distinct），72，293
distutils package（distutils 包），226
divide operation（触发运算），56
Djava.library.path=. Option（Djava.library.path=.项），237
DLL (Dynamic-link library), installing on Windows（在 Windows 上安装动态链接库），21－22，25－27
dlltool utility（dlltool 程序），22
DML (Data Manipulation Language)（数据控制语言），39，53，87
do() method（do()方法），223
Documentation component（文档部分），282
domain integrity（域完整性），97－101
 check constraints（check 约束），99－100
 and collations（排序规则），101
 default values（默认值），98
 foreign key constraints（外键约束），100－101
 NOT NULL constraints（非空约束），98－99
double data type（double 数据类型），241
downloading SQLite（下载 SQLite），17－18
dpkg utility（dpkg 程序），31
DROP COLUMN command（删除列命令），12
drop index command（删除索引命令），107，193
drop table command（删除表命令），193
drop table statement（删除表语句），286
DROP VIEW command（删除视图命令），106

dselect utility（dselect 程序），31
DSN (data source name)（数据源名称），248
.dump command（.dump 命令），39－40，42－43
dylib file（dylib 文件），262
.dylib shared library（共享库.dylib），17
Dynamic-link library (DLL), installing on Windows（在 Windows 上安装动态链接库），21－22，25－27

■E

.e command（.e 命令），34
echo() function（echo()函数），203－204，233
.echo command（.echo 命令），40
Edit System Variables dialog box(编辑系统变量对话框），19
else condition（else 情况），84
embedded databases（嵌入式数据库），1
Empty Project check box（空白工程复选框），26，28
endTransaction() method(endTransaction()方法），289
entity integrity（实体完整性），93－97
 primary key constraints（主键约束），94－97
 unique constraints（唯一性约束），93
Environmental Variables button（环境变量按钮），19
episodes table（episodes 表），7，48，78－79，145－146，151，306
episodes_foods table（episodes_foods 表），78
equality (=) operator（等于操作符），108
errmsg argument（参数 errmsg），157
error conditions, for functions in Extension C API（扩展 C API 中的错误情况函数），203
error handling in C API（C API 中的错误处理），174－176
 of busy conditions（繁忙情况），176－177
 of schema changes（模式改变），177－178
 and SQLite API（SQLite API），133－134
 with triggers（触发器），110
eTextRep argument（参数 eTextRep），197
example database（示例数据库），47－50
 installation（安装），48－49
 running examples（运行示例），49－50
examples folder（examples 文件夹），195
examples zip file（examples zip 文件），221
except keyword（关键字 except），81
except operation（except 操作），82
exclusive command（exclusive 命令），118
EXCLUSIVE lock（独占锁），138，143－145，148
exclusive state（独占状态）
 and transactions（事务），145
 for write transactions（写操作事务），142－143
exclusive transaction type（独占事务类型），117－118
exec() function, in C API（C API 中的 exec() 函数），155－159
exec_query() function（exec_query() 函数），240
execSQL() methods (execSQL()方法），287－288
execute() method（execute()方法），222－224，228－229，233
executemany() method(executemany()方法），230
.exit command（.exit 命令），20，32，34，37，42
explain query plan command（查询计划解释命令），123
exporting data from database（从数据库导出数据），39－41
Extension C API（扩展 C API），195－217
 aggregates with（聚合），204－209
 example of（示例），206－209
 registering（注册），205－206
 collations with（排序规则），209－217

defined（定义），210 - 212
on demand（随需而应），216 - 217
example of（示例），212 - 216
overview of（概要），210 - 212
types of（类型），212
functions with（函数），200 - 204
cleanup handlers for（清理处理），202 - 203
error conditions for（错误情况），203
returning input values（返回输入值），2lues，203 - 204
values for（值），202
registering functions in（注册函数），196 - 197
step function for（step 函数），198
values for（值），198 - 199

■F

f parameter（参数 f），266
factory parameter（参数 factory），287
fail resolution（失败处理），113 - 114
fcntl() implementation（fcntl()函数的实现），12
features（特性），8 - 11
compactness（紧凑性），9
convenience（便捷性），10 - 11
flexibility（灵活性），9
liberal licensing（自由授权），9
portability（可移植性），8
reliability（可靠性），10
simplicity（简单性），9
zero configuration（零配置），8
fetch() method（fetch()方法），222 - 223
fetchrow_array() method（fetchrow_array()方法），223
fetchrow_hashref() method（fetchrow_hashref()方法），223
f.id variable（变量 f.id），80
field types, and database file format（字段类型，数据库文件格式），305 - 306
file information, for database（数据库文件信息），44 - 45
file value（文件值），122
file.csv file（file.csv 文件），41
filename argument（参数 filename），154
filename parameter（参数 filename），118
file.sql file（file.sql 文件），39 - 40
fillData() method（fillData()方法），298
final() function（final()函数），198
finalize() method（finalize()方法），137，150 - 151，231
find utility（find 程序），3
finish() method（finish()方法），224
Fink package management system（Fink 包管理系统），31
flags parameter（参数 flags），155，287
flexibility of SQLite（SQLite 的灵活性），9
FLOAT class（FLOAT 类），196
float data type（浮点数据类型），241
FlushFileBuffers() function（FlushFileBuffers()函数），143
foo table（foo 表），118 - 119，185 - 186
foo_idx index（foo_idx 索引），108
Food class, in iSeinfeld example app（iSeinfeld 应用中的 Food 类），265 - 266
foo.db database（foo.db 数据库），11
FoodViewController class，in iSeinfeld example app（iSeinfeld 应用中的 FoodViewController 类），266 - 268
FoodViewController UIViewController class（FoodViewController UIViewController 类），266
FoodViewController.h file（FoodViewController.h 文件），266 - 267
FoodViewController.m file（FoodViewController.m 文件），267
FoodViewController.xib file（FoodViewController.xib

文件），266 - 267
FOR EACH ROW behavior（递归每一行），12
FOR EACH STATEMENT trigger（FOR EACH STATEMENT 触发器），12
foreign key，constraints of（外键约束），100 - 101
foreign key relationship（外键关系），72
fork() function（fork()函数），190
Formatting（格式化）
 data from database（来自数据库的数据），40 - 41
 SQL statements（SQL 语句），134 - 135
Fossil source control，compiling source code on Windows（Windows 上的 Fossil 源代码管理，编译源代码），23 - 25
fossil.exe file（fossil.exe 文件），23
framework alternatives，for iOS development（iOS 开发中可替换的框架），258 - 259
frequency column（frequency 列），84
frequency values（frequency 值），84
from clause（from 子句），57 - 59，67，70，73，80
frq argument（参数 frq），189
fsync() function（fsync()函数），143，320
full mode（全模式），122
full outer join（全外连接），76
FULL OUTER JOIN statement（全外连接语句），12
full synchronous pragma（编译指示 full synchronous），121
Functions（函数）
 for B-tree module API（B-tree 模块 API）
 access functions（访问函数），308 - 309
 configuration functions（配置函数），310
 cursor functions（游标函数），309
 record functions（记录函数），310
 table functions（表函数），309
 in Extension C API（扩展 C API 中），200 - 204
 cleanup handlers for（清理处理），202 - 203
 error conditions for（错误情况），203
 registering（注册），196 - 197
 returning input values（返回输入值），203 - 204
 values for（值），202
 SQL for SQLite（SQLite 中的 SQL），66
 user-defined（用户自定义），136 - 137

G

GCC (GNU Compiler Collection)（GNU 编译器集合），28，31
gdbm storage manager（gdbm 存储管理），4
get_table() function（get_table()函数），132 - 133，159 - 160
getAttribute() method（getAttribute()方法），PDO class（PDO 类），249
getReadableDatabase() method（getReadableDatabase()方法），285
getWriteableDatabase() method（getWriteableDatabase()方法），285
glob operator（glob 操作符），where clause（where 子句），64
GNU Compiler Collection (GCC)（GNU 编译器集合），28，31
GRANT command（授权命令），13
grep utility（grep 程序），3
group by clause（group by 子句），67 - 71
group_concat() function（group_concat()函数），206

H

.h command（.h 命令），34
.h file（.h 文件），26
Häing, Gerhard（PySQLiteV2 的创建者），226
having clause（having 子句），71
having predicate（having 断言），70

.headers command（.headers 命令），37－38，40
hello_newman() method（hello_newman()方法），136－137，200,225，231，236，240－241，247，251
.help command（.help 命令），20，32－33
-help switch（-help 转换），34
Hewlett-Packard Unix（惠普 Unix）(HP-UX)，3
Hipp, Richard（本书技术评审），3，5
history of SQLite（SQLite 的历史），3－4
hot journal（热日志），145
HP-UX (Hewlett-Packard Unix)（惠普 Unix），3
.htaccess controls（.htaccess 控制），248

I

ICL (imperative chef language)（命令式厨师语言），51
id column（id 列），37，54，76，87，89，94－95，110，124
id value（id 值），89，100
identifiers（标示符），53
if construct（if 结构体），271
ignore argument（参数 ignore），110
ignore resolution （ignore 解决办法），113
IllegalStateException exception（IllegalStateException 异常），289
immediate transaction type（立即事务类型），117－118
imperative chef language（命令式厨师语言）(ICL)，51
imperative language（命令式语言），50
.import [file][table] command（.import [file][table] 命令），40
.import command（.import 命令），39
importing data into database（将数据导入到数据库），40
IN operator（IN 操作符），61，79，108
index_info schema pragma（编译指示 index_info schema），121
index_list schema pragma（编译指示 index_list schema），121
index_name variable（变量 index_name），106
Indexes（索引），106－108
 and collations for columns(列的排序规则)，107
 utilization of（利用），107－108
.indices [table name] command（.indices [table name]命令），37
.indices shell command（.indices shell 命令），107
init function（init 函数），226，265
init option（init 项），42
initialValues ContentValue map（initialValues ContentValue 映射），289
initialValues map（initialValues 映射），289
initWithName function（initWithName 函数），265－266
inline views（内嵌视图），80
inner joins（内连接)，74－75
Input submenu（输入子菜单），26，28
insert() method（insert()方法），289
insert command（插入命令），87－91
 multiple rows（多行），90－91
 one row（一行），87－89
 set of rows（一组），89－90
insert filter（插入过滤），183
insert operation（插入操作），135，179－180
insert or replace expression（插入或替代表达式），114
INSERT OR REPLACE (...) statement（插入或替代语句），10
insert statements（插入语句），37，88－89，98，132，159,183，288，312，315
insert trigger（插入触发器），109，111
Install New Software screen（安装新软件界面），280
Installation Type screen（安装类型选择界面），256

instead of keywords（关键字 instead of），110
INSTEAD OF triggers（INSTEAD OF 触发器），12
int data type（整型数据类型），241
integer class（integer 类），102－103，196
integer column（整型类），314
integer primary key（整型主键），54，88，94－96
integer type（整型类型），54
integrity_check pragma（编译指示 integrity_check），122
interface，architecture of SQLite（SQLite 的接口，结构），6
Interface Builder tool（接口构建工具），267－268
intersect keyword（关键字 intersect），81
intersect operation（intersect 操作），81－82
Intersection（交集），74
inTransaction() method（inTransaction()方法），289
iOS development（iOS 开发），253－277
 framework alternatives for（可替换的框架），258－259
 iSeinfeld example app（iSeinfeld 应用），259－276
 accessing database upon startup（启动时访问数据库），269－272
 adding SQLite framework to project（向工程添加 SQLite 框架），261－262
 creating Xcode project（创建 Xcode 工程），259－260
 Food class in（Food 类），265－266
 foods database or（foods 数据库），263－264
 FoodViewController class in（FoodViewController 类），266－268
 and large databases（大数据库），276
 prerequisites for（必备条件），253－259
 Developer Program membership（开发者计划资格），254
 downloading iOS SDK（下载 iOS SDK），254－258
is not null operator（is not null 操作符），85
is null operator（is null 操作符），85
IS operator（IS 操作符），62
iSeinfeld example app（iSeinfeld 应用），259－276
 accessing database upon startup（启动时访问数据库），269－272
 adding SQLite framework to project（向工程添加 SQLite 框架），261－262
 creating Xcode project（创建 Xcode 工程），259－260
 Food class in（Food 类），265－266
 foods database or（foods 数据库），263－264
 FoodViewController class in（FoodViewController 类），266－268
iSeinfeldAppDelegate class（iSeinfeldAppDelegate 类）269
iSeinfeldAppDelegate.h file（iSeinfeldAppDelegate.h 文件），269
iSeinfeldAppDelegate.m file（iSeinfeldAppDelegate.m 文件），269－270，274
isReadOnly() method（isReadOnly()方法），285－286

■J

Java，language extension for（语言扩展），236－243
 connecting to（连接），238
 installation of（安装），237
 JDBC support with（JDBC 支持），241－243
 query processing with（查询处理），238－240
 user-defined functions with（用户自定义函数），240－241
Java Database Connectivity（Java 数据库连接）(JDBC), support for Java language

extension（java 语言扩展对 JDBC 的支持），241 - 243
Java Development Kit（Java 开发套件）(JDK)，280
Java runtime environment（Java 运行环境）(JRE)，280
java.lang.String data type（java.lang.String 数据类型），241
java.sql.Date data type）（java.sql.Date 数据类型），241
javasqlite3.mak file（javasqlite3.mak 文件），237
javasqlite.mak file（javasqlite.mak 文件），237
java.sql.Time data type（java.sql.Time 数据类型），241
java.sql.Timestamp data type（java.sql.Timestamp 数据类型），241
JDBC (Java Database Connectivity)（Java 数据库连接），support for Java language extension（Java 语言扩展对 JDBC 的支持），241 - 243
JDK （Java 开发套件)，280
join condition（连接条件），75
joining tables（连接表），72 - 77
 cross joins（交叉连接），75 - 76
 inner joins（内连接)，74 - 75
 natural joins（自然连接），77
 outer joins（外连接），76
 preferred syntax for（优先语法），77
journal_mode PRAGMA（编译指示日志模式），320
JRE (Java runtime environment)（Java 运行环境），280
justincase savepoint（以防万一保存点），112

K

keywords（关键字），53

L

Label elements（标签元素），267
language extensions（扩展语言），219 - 252
 Java，236 - 243
 connecting to（连接），238
 installation of（安装），237
 JDBC support with（JDBC 支持），241 - 243
 query processing with（查询处理），238 - 240
 user-defined functions with（用户自定义函数），240 - 241
 Perl，221 - 226
 aggregates with（聚合），225 - 226
 connecting to（连接），222
 installation of（安装），221 - 222
 parameters for（参数），224
 query processing with（查询处理），222 - 224
 user-defined functions with（用户自定义函数），224 - 225
 PHP，247 - 252
 connecting to（连接），248
 installation of（安装），248
 query processing with（查询处理），248 - 251
 user-defined functions with（用户自定义函数），251 - 252
 Python，226 - 232
 aggregates with（聚合），231 - 232
 APSW as alternative（另一种选择 APSW），232
 connecting to（连接），227
 installation of（安装），226 - 227
 parameters for（参数），229 - 230
 query processing with（查询处理），227 - 229
 user-defined functions with（用户自定义函数），231
 Ruby，232 - 236
 connecting to（连接），233
 installation of（安装），232 - 233

parameters for（参数），234 - 235
query processing with（查询处理），233 - 234
user-defined functions with（用户自定义函数），236
selecting（选择），220 - 221
Tcl，243 - 247
connecting to（连接），244
installation of（安装），243 - 244
query processing with（查询处理），244 - 246
user-defined functions with（用户自定义函数），247
large databases（大数据库），and Android Development（Android 开发），300 - 301
large file support (LFS)（大文件支持），176
last_insert_rowid() function（last_insert_rowid() 函数），37，88 - 89，159
last_step() function（last_step()函数），241
left outer join（左外连接），76
LEFT OUTER JOIN statement（左外连接语句），12
length_first collation（长度优先排序规则），212 - 214，216
length(name) expression（取长度表达式），67
LFS (large file support)（大文件支持），176
liberal licensing，9features of SQLite（SQLite 的自由授权特性），9
libsqlite_jni.so file（libsqlite_jni.so 文件），237
libsqlite3.0.dylib file（libsqlite3.0.dylib 文件），262
libsqlite3.dylib library（libsqlite3.dylib 库文件），262
life cycles，for transactions（事务的生命周期），138 - 139
like operator（like 操作符），37，62 - 64
limit clause（limit 子句），64，80，293
limit keyword（关键字 limit），64 - 65
Linker folder（Linker 文件夹），26，28
Linux，30 - 32
binaries（二进制包）and packages（包），30 - 31
compiling SQLite from source（从源代码编译），31 - 32
.list command（.list 命令），40
literals，syntax for SQL in SQLite（SQLite 中的字符，SQL 语法），52
lock states（锁定状态），transactions（事务），139 - 141
LockFile() function（LockFile()函数），148
LockFileEx() function（LockFileEx()函数），148
locks（锁）
and deadlocks（死锁），116 - 117
and transactions（事务），115 - 116
log table（日志表），109
log_sql() function（log_sql()函数），201
logical operators（操作符），for where clause（where 子句），63
lower() function（lower()函数），66

M

Mac OS X（苹果 Mac OS X 系统），30 - 32
binaries（二进制包）and packages（包），30 - 31
compiling SQLite from source（从源代码编译），31 - 32
MacPorts package management system（MacPorts 包管理系统），31
main database（主数据库），119
main.xml file（main.xml 文件），297
make command（make 命令），30
manifest typing（显示类型），311 - 313
match predicate（match 断言），64
max() function（max()函数），67
maxed_out table（maxed_out 表），96
maxed_out.id value（maxed_out.id 值），95
Maximum VM application heap size option（虚拟

机应用程序最大堆栈大小），284
memcmp() function（memcmp()函数），101，103－104，195,212
memory management，and threading support in C API（C API 中的内存管理，线程支持），193
:memory: string（:memory:字符串），128
memory value（内存值），122
MERGE statement（合并语句），10
methods，for SQLiteDatabase class（SQLiteDatabase 类的方法），288－290
min() function（min()函数），67
MinGW，building SQLite with on Windows（使用 MinGW 在 Windows 上构建 SQLite），28－30
.mode command（.mode 命令），37－38，40
modifying data（修改数据），87－92
Module Definition File property page（模块定义文件属性页面），26
multiple connections，using in code（代码中使用多个连接），149－150
myDatabaseAdapter class（myDatabaseAdapter 类），298
myDatabaseHelper class（ myDatabaseHelper 类），294，296，300

■N

n parameter（参数 n），266
name attribute（name 属性），48，106
name column（name 列），54－55，87－88，93，98，101,109，115，123
name parameter（参数 name），104，106，285
name table（name 表），104
name value（name 值），101，109
named parameters，in C API（C API 中的命名参数），173
nArg argument（参数 nArg），197－198
natural joins（自然连接），77
nBytes argument（参数 nBytes），198
new.name column（new.name 列），110
newrow() method（newrow()方法），239
next() method（next()方法），233
no action action（no action 行为），101
nocase collation（大小写不敏感排序规则），101，138，212
normal synchronous pragma（编译指示 normal synchronous），122
not null constraints（非空约束），54，98－99，113
NOT operator（NOT 操作符），64
null（空），84
NULL class（NULL 类），102－103，196
null values（null 值），83－84，89
nullColumnHack parameter（参数 nullColumnHack），289
nullif function（.nullif 函数），86
.nullvalue command（.nullvalue 命令），40
numbered parameters，in C API（C API 中的编号参数），172

■O

ODBC (Open Database Connectivity)（开放数据库互连），4
off synchronous pragma（编译指示 off synchronous），122
offset clause（offset 子句），64
offset keyword（关键字 offset），64－65
oid value（oid 值），94
on delete restrict option（删除约束项），100
onCreate() method（onCreate()方法），285－286，298
onOpen() method（onOpen()方法），285－286
onUpgrade() method（onUpgrade()方法），285－286
open command（open 命令），25

Open Database Connectivity (ODBC)（开放数据库互连），4
Open drop-down box（打开下拉选项框），19
openDatabase() method（openDatabase()方法），287
opening databases，with SQLiteDatabase class（用SQLiteDatabase 打开数据库），286 - 287
openOrCreate Database() method（openOrCreate Database()方法），287
operation_code argument（参数 operation_code），180
operational control，with SQLite API（SQLite API 操作控制），135
operators（操作符）
 glob operator（glob 操作符），where clause（where 子句），64
 for where clause（where 子句），60 - 63
 binary operators（二进制操作符），60 - 62
 like operator（like 操作符），63
 logical operators（逻辑操作符），63
 OR operator（OR 操作符），61
order by clause（order by 子句），64 - 66，80 - 82，210
order by keyword（关键字 order by），66
order by subquery（order by 子查询），80
outer joins（外连接），76
.output [filename] command（.output [filename]命令），39
.output stdout command（.output stdout 命令），39
overflow pages，and database file format（溢出页，数据库文件格式），307 - 308
Owens，Michael（本书作者），226

■P

p->zResult field（p->zResult 域），208
package require directive（包需求指令），244
packages（包）
 for Linux（Linux 系统），30 - 31
 for Mac OS X（苹果 Mac OS X），30 - 31
 for POSIX systems（POSIX 系统），30 - 31
page cache（页面缓存），and write transactions（写操作事务），145 - 146
page_size pragma（编译指示 page_size），128
pager（SQLite 中的一种数据结构），127
pages，and database file format（页面，数据库文件信息）
 overflow pages in（溢出页），307 - 308
 reuse of（重用页），304
parameters
 in C API（C API 中的参数），169 - 174
 named parameters（命名参数），173
 numbered parameters（编号参数），172
 Tcl parameters（Tcl 参数），173 - 174
 for language extensions（扩展语言）
 for Perl（Perl），224
 for Python（Python），229 - 230
 for Ruby（Ruby），234 - 235
 and SQLite API（SQLite API），131 - 132
Path entry（Path 项），19
PATH environment variable（环境变量 PATH），23
pBlocked parameter（参数 pBlocked），193
PDO (PHP Data Objects)（PHP 数据对象），247 - 248
PDO_ATTR_TIMEOUT parameter（参数 PDO_ATTR_TIMEOUT），249
PDOStatement class（PDOStatement 类），248
PDOStatement::bindColumn() method（PDOStatement::bindColumn()方法），250
PDOStatement::bindParam() method（PDOStatement::bindParam()方法），249
pending state（待定状态），for write transactions（写操作事务），142
performance（性能）

and limitations of SQLite（SQLite 局限），11 - 13
and WAL（预写日志），321
Perl，language extension for（Perl 语言扩展），221 - 226
aggregates with（聚合），225 - 226
connecting to（连接），222
installation of（安装），221 - 222
parameters for（参数），224
query processing with（查询处理），222 - 224
user-defined functions with（用户自定义函数），224 - 225
perlsum() aggregate（perlsum()聚合），225
perlsum.pm package（perlsum.pm 包），225
phone column（phone 列），54，93，98 - 99
phone value（phone 值），93，99，101
PhoneGap framework（PhoneGap 框架），258，284
PHP，language extension for（PHP 语言扩展），247 - 252
connecting to（连接），248
installation of（安装），248
query processing with（查询处理），248 - 251
user-defined functions with（用户自定义函数），251 - 252
PHP Data Objects (PDO)（PHP 数据对象），247 - 248
phpsum aggregate（phpsum 聚合），251
Platforms component，Android（Android 平台组件），282
pNotifyArg pointer（指针 pNotifyArg），192
portability，features of SQLite（SQLite 的可移植性），8
POSIX systems（POSIX 系统），30 - 32
binaries（二进制包）and packages（包），30 - 31
compiling SQLite from source（从源代码编译），31 - 32
power consumption（功耗），12
ppDb argument（参数 ppDb），155
ppStmt parameter（参数 ppStmt），161
PRAGMA commands（PRAGMA 命令），39
PRAGMA journal_mode=WAL setting（日志模式 journal_mode=WAL 设置），319
pragmas（编译指示），120 - 122
auto_vacuum（自动清理），122
cache size（缓存大小），120
database schema（数据库模式），120 - 121
integrity_check（完整性检查），122
synchronous（同步），121 - 122
temp_store（临时存储），122
temp_store_directory（临时存储目录），122
Precompiled binaries For Windows section（Windows 上二进制包），21
--prefix=DIR option（--prefix=DIR 项），237
prepare() method（prepare()方法），131 - 132，222，224，249
prepared queries
with C API（C API 准备查询），161 - 164
compilation of（编译），161 - 162
execution of（执行），162
finalization of（结束），163 - 164
executing（执行），129 - 131
prerequisites
for Android development（Android 开发的必备条件），279 - 284
additional components（额外组件），281 - 284
downloading Android Developer Tools（下载 Android 开发工具），280 - 281
downloading Android SDK Starter Package（下载 Android SDK Starte 包），280
JDK（Java 开发套件），280
for iOS development（iOS 开发），253 - 259
Developer Program membership（开发者计

划资格），254
downloading iOS SDK（下载 iOS SDK），254 - 258
primary key constraints（主键约束），54，94 - 97
print_sql_result() function（print_sql_result()函数），187，201
.prompt [value] command（.prompt [value]命令），40
Property Pages dialog box（属性页对话框），26
public boolean isDbLockedByOtherThreads() method（public boolean isDbLockedByOtherThreads()方法），290
public int getVersion() method（public int getVersion()方法），290
public long getMaximumSize() method（public long getMaximumSize()方法），290
public static int releaseMemory() method（public static int releaseMemory()方法），290
pUserdata argument（参数 pUserdata），197 - 198
pysum() aggregate（pysum()聚合），137，231
Python，language extension for（Python 语言扩展），226 - 232
aggregates with（聚合），231 - 232
APSW as alternative（另一种选择 APSW），232
connecting to（连接），227
installation of（安装），226 - 227
parameters for（参数），229 - 230
query processing with（查询处理），227 - 229
user-defined functions with（用户自定义函数），231
pzTail parameter（参数 pzTail），164

■Q

queries, with SQLiteDatabase class（用 SQLiteDatabase 类查询），287 - 288
query() method（query()方法），239，248 - 249，287 - 288
querying databases（查询数据库），55 - 86
compound queries（组合查询），81 - 83
executing prepared queries（执行准备查询），129 - 131
executing wrapped queries（执行封装查询），132 - 133
relational operations（关系操作），55 - 56
subqueries（子查询），79 - 80
queryWithFactory() method（queryWithFactory()方法），288

■R

raise() function（raise()函数），110
RANK() function（RANK()函数），13
rawQuery() method（rawQuery()方法），287 - 288，294
rawQueryWithFactory() method（rawQueryWithFactory()方法），288
RDBMS (relational database management system)（关系数据库管理系统），1
.read command（.read 命令），40
read transactions（读操作事务），141
read_committed pragma（编译指示 read_committed），191
read_uncommitted pragma（编译指示 read_uncommitted），151，191
readFoodsFromDatabase function（readFoodsFromDatabase 函数），270
README file（README 文件），196
read-uncommitted isolation level（读取未提交隔离级别），shared cache mode（共享缓存模式），191 - 192
read-uncommitted mode（读取未提交模式），151
real class（real 类），102 - 103
record functions，for B-tree module API（B-tree

模块 API 记录函数），310
records，and database file format（记录与数据库文件格式），304－305
recovery，issues with WAL（WAL 的恢复问题），321－322
recursive triggers（递归触发器），12
referential integrity（引用完整性），92
regexp predicate（正则表达式断言），64
registering in extension C API（扩展 C API 中的注册）
 aggregates（聚合），205－206
 functions（函数），196－197
reindex command（重建索引命令），119
relational database management system (RDBMS)（关系数据库管理系统），1
relational operations，querying databases（关系操作，查询数据库），55－56
relational operators（关系操作符），62
release command（release 命令），112
reliability，features of SQLite（SQLite 的可靠性），10
rename clause（rename 子句），55
rename operation（重命名操作），77
RENAME TABLE command（RENAME TABLE 命令），12
replace() methods（replace()方法），289
replace command（replace 命令），114
res folder（res 文件夹），297
RESERVED connection（保留连接），148
RESERVED lock（保留锁），140－141，144，148
reserved state（保留状态），140－142，144－145，147－148
reset() function（reset()函数），132，150－151
Resources folder（Resources 文件夹），263
restrict action（restrict 行为），101
resultp argument（参数 resultp），159
resultp pointer（指针 resultp），159
reverse collation（反向排序规则），101
REVOKE command（REVOKE 命令），13
right outer join（右外连接），76
RIGHT OUTER JOIN statement（右外连接语句），12
R.java file（R.java 文件），298
rollback() method（rollback()方法），249
rollback command（回滚命令），111－112，139
rollback resolution（回滚解决方法），113，115
rollback_hook() function，in C API（C API 中 rollback_hook()函数），179
rootpage column（rootpage 列），123
Row class（Row 类），239
ROW_NUMBER() function（ROW_NUMBER()函数），13
rowid argument（参数 rowid），180
rowid column（rowid 列），96－97
rowid value（rowid 值），94－96
rows，and insert command（行，插入命令）
 multiple rows（多行），90－91
 one row（一行），87－89
 set of rows（一组），89－90
rsync utility（rsync 程序），3
RTRIM collation（RTRIM 排序规则），138
Ruby，language extension for（Ruby 语言扩展），232－236
 connecting to（连接），233
 installation of（安装），232－233
 parameters for（参数），234－235
 query processing with（查询处理），233－234
 user-defined functions with（用户自定义函数），236

■S

SAggCtx struct（SAggCtx 结构体），208
Samples component（示例部分），282
savepoint command（保存点命令），112

.schema [table name] command（.schema [table name]命令），38
.schema command（.schema 命令），42，55
.schema foods shell command（.schema foods shell 命令），88
schema information，administration of databases（数据库的模式信息与管理），37－39
.schema shell command（.schema shell 命令），55
schema view（模式视图），37，40
scope，for transactions（事务范围），111－112
SDK Starter Package(SDK Starter 包)，prerequisite for Android development（Android 开发的必备条件），280
Seinfeld Android app（Seinfeld Android 应用），294－300
 adding database to project（将数据库添加到工程），296
 creating project（创建工程），295－296
 linking data and user interface(连接数据与用户接口），298－299
 querying foods table(查询 foods 表)，296－297
 user interface for（用户接口），297－298
 viewing app（查看应用），299－300
select clause（select 子句），57－59，65，67－68，70－71，79，89
select command（选择命令），55－59，67，72，75，80，87
select form（从...选择），insert command（插入命令），90
select hello_newman() function（选择 hello_newman()函数），137
select id clause（select id 子句），77
SELECT statements（SELECT 语句），10－11，129，131，210
selectrow() function（selectrow()函数），222
select-stmt parameter（参数 select-stmt），104
.separator command（.separator 命令），40
SET clause（SET 子句），92
set default action（设置默认行为），101
set null action（设置空行为），101
set_authorizer() function，in C API（C API 中的 set_authorizer()函数），180－189
setAttribute() method（setAttribute()方法），PDO class（PDO 类），249
setDistinct() method（setDistinct()方法），293
setProjectionMap (Map<String，String> columnMap) method（etProjectionMap (Map<String，String>columnMap 方法），293
setTables() method（setTables()方法），293
setTransactionSuccessful() method（setTransactionSuccessful()方法），289
setup.cfg file（setup.cfg 文件），227
setup.exe file（setup.exe 文件），28－29
shared cache mode（共享缓存模式）
 overview（纵览），151－152
 threading support in C API（C API 中的线程支持），190－193
 read-uncommitted isolation level（读取未提交隔离级别），191－192
 unlock notification（解锁通知），192－193
SHARED connection（共享连接），148
Shared library（共享库），31
SHARED locks（共享锁），142，144，146－147，149－151
shared object (so)（共享对象），17
SHARED state（共享状态），140－142，144－145
shell mode，command-line program in（shell 模式，命令行程序），33－34
shell.c file（shell.c 文件），26，28
short data type（short 数据类型），241
.show command（.show 命令），40
show_datatypes pragma（编译指示 show_datatypes），242

simplicity, features of SQLite（SQLite 的简单性），9
sleep() function（sleep()函数），177
so (shared object)（共享对象），17
source clause（source 子句），78
source code（源代码）
 compiling on Linux（在 Linux 上编译），31 - 32
 compiling on Mac OS X（在 Mac OS X 上编译），31 - 32
 compiling on POSIX systems（在 POSIX 系统上编译），31 - 32
 compiling on Windows（在 Windows 上编译），22 - 25
 Fossil source control（Fossil 源代码管理），23 - 25
 stable source distribution（稳定的源代码发布版），22
Source Code section（源代码一节），22
spam_mother_in_law() function（spam_mother_in_law()函数），147
sprintf() function（sprintf()函数），134
sql argument（参数 sql），156，159
sql column（sql 列），123
SQL for SQLite（SQLite 的 SQL）
 administration（管理），118 - 124
 attaching databases（附着数据库），118 - 119
 cleaning databases（清理数据库），119
 explain query plan command（查询计划解释命令），123
 and pragmas（编译指示），120 - 122
 of pragmas（编译指示），122
 sqlite_master table（sqlite_master 表），123
 aggregates（聚合），67
 aliases，for tables（表的别名），77 - 79
 and ambiguity of column names（列名称的歧义性），77
 case expression（case 表达式），83 - 84
 compound queries（组合查询），81 - 83
 creating databases（创建数据库），53 - 55
 altering tables（修改表），54 - 55
 creating tables（创建表），53 - 54
 data integrity（数据完整性），92 - 104
 domain integrity（域完整性），97 - 101
 entity integrity（ 实体完整性），93 - 97
 indexes（索引），106 - 108
 storage classes for SQLite（SQLite 存储类），101 - 104
 triggers（触发器），108 - 111
 views in SQLite（SQLite 中的视图），104 - 106
 delete command（删除命令），92
 distinct clause（distinct 子句），72
 example database（示例数据库），47 - 50
 installation（安装），48 - 49
 running examples（运行示例），49 - 50
 functions（函数），66
 group by clause（group by 子句），67 - 71
 Indexes（索引），106 - 108
 and collations for columns（列的排序规则），107
 utilization of（利用），107 - 108
 insert command（插入命令），87 - 91
 multiple rows（多行），90 - 91
 one row（一行），87 - 89
 set of rows（一组），89 - 90
 joining tables（连接表），72 - 77
 cross joins（交叉连接），75 - 76
 inner joins（内连接），74 - 75
 natural joins（自然连接），77
 outer joins（外连接），76
 preferred syntax for（优先语法），77
 limit clause（limit 子句），64
 modifying data（修改数据），87 - 92

null in SQLite（SQLite 中的 null），84
order by clause（order by 子句），64 - 66
querying databases（查询数据库），55 - 86
select command（选择命令），57 - 59
storage classes for（存储类），101 - 104
subqueries（子查询），79 - 80
syntax for（语法），50 - 53
commands（命令），51 - 52
comments（注释），53
identifiers（标示符），53
keywords（关键字），53
literals（字符），52
transactions（事务），111 - 118
and conflict resolution（冲突解决），112 - 115
and deadlocks（死锁），116 - 117
and locks（锁），115 - 116
scope for（范围），111 - 112
types of（类型），117 - 118
triggers（触发器），108 - 111
error handling with（错误处理），110
and updatable views（可更新的视图），110 - 111
update triggers（更新触发器），109 - 110
update command（更新命令），91 - 92
views in（其中的视图），104 - 106
where clause（where 子句），59 - 64
binary operators（二进制操作符），60 - 62
glob operator（glob 操作符），64
like operator（like 操作符），63
logical operators（逻辑操作符），63
operators（操作符），60
values（值），60
SQLite
for administrators（管理员），3
for developers（开发者），2 - 3
downloading（下载），17 - 18
embedded database（嵌入式数据库），1
history of（历史），3 - 4
tools for（工具），45
where used（用处），4 - 5
SQLite API，125 - 138
core API（核心 API），127 - 135
connecting to database（连接数据库），128
and error handling（错误处理），133 - 134
executing prepared queries（执行准备查询），129 - 131
executing wrapped queries（执行封装查询），132 - 133
formatting SQL statements（格式化 SQL 语句），134 - 135
and parameters（参数），131 - 132
operational control with（操作控制），135
principal data structures（主要数据结构），126 - 135
B-tree and pager（B-tree 与 pager），127
connections and statements（连接与语句），126 - 127
user-defined extensions（用户自定义扩展）
aggregates（聚合），137
collations（排序规则），138
functions（函数），136 - 137
using threads（使用线程），136
sqlite directory（sqlite 历史），33
sqlite_analyzer view（sqlite_analyzer 视图），44
SQLITE_ATTACH event（ATTACH 事件），182
SQLITE_BUSY error（SQLITE_BUSY 错误），118，133
SQLITE_BUSY value（SQLITE_BUSY 值），146 - 148，151
sqlite_column_type() function（sqlite_column_type()函数），199
sqlite_complete() function（sqlite_complete()函数），164
sqlite_create_funtion() function（sqlite_create_

funtion()函数），64
SQLITE_CREATE_INDEX event
（SQLITE_CREATE_INDEX 事件），181
SQLITE_CREATE_TABLE event
（SQLITE_CREATE_TABLE 事件），181
SQLITE_CREATE_TEMP_INDEX event
（SQLITE_CREATE_TEMP_INDEX 事件），181
SQLITE_CREATE_TEMP_TABLE event
（SQLITE_CREATE_TEMP_TABLE 事件），181
SQLITE_CREATE_TEMP_TRIGGER event
（SQLITE_CREATE_TEMP_TRIGGER 事件），181
SQLITE_CREATE_TEMP_VIEW event
（SQLITE_CREATE_TEMP_VIEW 事件），181
SQLITE_CREATE_TRIGGER event
（SQLITE_CREATE_TRIGGER 事件），181
SQLITE_CREATE_VIEW event
（SQLITE_CREATE_VIEW 事件），181
SQLITE_DELETE event（SQLITE_DELETE 事件），181
SQLITE_DENY constant（常量 SQLITE_DENY），180
SQLITE_DETACH event（SQLITE_DETACH 事件），182
SQLITE_DONE value（SQLITE_DONE 值），129，151
SQLITE_DROP_INDEX event
（SQLITE_DROP_INDEX 事件），181
SQLITE_DROP_TABLE event
（SQLITE_DROP_TABLE 事件），181
SQLITE_DROP_TEMP_INDEX event
（SQLITE_DROP_TEMP_INDEX 事件），181
SQLITE_DROP_TEMP_TABLE event（事件），181
SQLITE_DROP_TEMP_TRIGGER event
（SQLITE_DROP_TEMP_TRIGGER 事件），181
SQLITE_DROP_TEMP_VIEW event
（SQLITE_DROP_TEMP_VIEW 事件），181
SQLITE_DROP_TRIGGER event
（SQLITE_DROP_TRIGGER 事件），181
SQLITE_DROP_VIEW event
（SQLITE_DROP_VIEW 事件），181
SQLITE_ENABLE_COLUMN_METADATA directive
（SQLITE_ENABLE_COLUMN_METADATA 指令），166
SQLITE_ENABLE_MEMORY_MANAGEMENT directive
（SQLITE_ENABLE_MEMORY_MANAGEMENT 指令），193
SQLITE_ERROR error（SQLITE_ERROR 错误），133
SQLITE_FULL error（SQLITE_FULL 错误），95
SQLITE_IGNORE constant（常量 SQLITE_IGNORE），180，183
SQLITE_INSERT event（SQLITE_INSERT 事件），181
SQLITE_LOCKED value（SQLITE_LOCKED 值），151
sqlite_master table（sqlite_master 表），107，123，186，188，192，303－304
sqlite_master view（sqlite_master 视图），38，44，135
SQLITE_NOMEM error（SQLITE_NOMEM 错误），208
SQLITE_OK constant（常量 SQLITE_OK），180
SQLITE_OPEN_CREATE flag
（SQLITE_OPEN_CREATE 标志），154

SQLITE_OPEN_FULLMUTEX flag（SQLITE_OPEN_FULLMUTEX 标志），155
SQLITE_OPEN_NOMUTEX flag（SQLITE_OPEN_NOMUTEX 标志），155
SQLITE_OPEN_PRIVATECACHE flag（SQLITE_OPEN_PRIVATECACHE 标志），155
SQLITE_OPEN_SHAREDCACHE flag（SQLITE_OPEN_SHAREDCACHE 标志），155
SQLITE_PRAGMA event（SQLITE_PRAGMA 事件），181
SQLITE_READ event（SQLITE_READ 事件），180－181，187
SQLITE_RESULT value（SQLITE_RESULT 值），151
sqlite_result_xxx() functions（sqlite_result_xxx() 系列函数），202
SQLITE_ROW value（SQLITE_ROW 值），129
SQLITE_SCHEMA condition（SQLITE_SCHEMA 情况），178，189
SQLITE_SCHEMA errors（SQLITE_SCHEMA 错误），162，178，192
SQLITE_SCHEMA event（SQLITE_SCHEMA 事件），189
SQLITE_SELECT event（SQLITE_SELECT 事件），181，187
sqlite_sequence table（sqlite_sequence 表），95
SQLITE_TRANSACTION event（SQLITE_TRANSACTION 事件），182
SQLITE_UPDATE event（SQLITE_UPDATE 事件），180，182
sqlite3 command（sqlite3 命令），244
sqlite3 handle（sqlite3 句柄），126，128，132
sqlite3 package（sqlite3 包），244
sqlite3 structure（sqlite3 结构），248
sqlite3_aggregate_context() function（sqlite3_aggregate_context()函数），198，207－208
sqlite3_analyzer tool（sqlite3_analyzer 工具），145
sqlite3_bind() function（sqlite3_bind()函数），176
sqlite3_bind_double() function（sqlite3_bind_double()函数），170
sqlite3_bind_parameter_index() function（sqlite3_bind_parameter_index()函数），173
sqlite3_bind_xxx() functions（sqlite3_bind_xxx() 系列函数），167，170，202
sqlite3_busy_handler() function（sqlite3_busy_handler（函数），176－177
sqlite3_busy_timeout() function（sqlite3_busy_timeout()函数），147，176－177
sqlite3_changes() function（sqlite3_changes()函数），159
sqlite3_close() function（sqlite3_close()函数），155，192，272
sqlite3_collation_needed() function（sqlite3_collation_needed()函数），216
sqlite3_column_blob() function（sqlite3_column_blob()函数），167
sqlite3_column_bytes() function（sqlite3_column_bytes()函数），167
sqlite3_column_count() function（sqlite3_column_count()函数），165
sqlite3_column_decltype() function（sqlite3_column_decltype()函数），165－166
sqlite3_column_name() function（sqlite3_column_name()函数），165
sqlite3_column_origin_name() function（sqlite3_column_origin_name()函数），

166
sqlite3_column_text() method
（sqlite3_column_text()方法），272
sqlite3_column_type() function
（sqlite3_column_type()函数），165
sqlite3_column_xxx() functions
（sqlite3_column_xxx()系列函数），161，
166,199，203，272
sqlite3_commit_hook() function
（sqlite3_commit_hook()函数），135，
178－179
sqlite3_complete() function（sqlite3_complete()
函数），164
sqlite3_create_collation() function
（sqlite3_create_collation()函数），138
sqlite3_create_collation_v2() function（sqlite3_
create_collation_v2()函数），213
sqlite3_create_function() function
（sqlite3_create_function()函数），
136,196－198，205，213
sqlite3_data_count() function（sqlite3_data_count()
函数），165
sqlite3_db_handle() function（sqlite3_db_handle()
函数），169
sqlite3_errcode() function（sqlite3_errcode()函
数），134
sqlite3_errmsg() function（sqlite3_errmsg()函数），
134，162，169,174
sqlite3_exec() function（sqlite3_exec()函数），
132，155－156，158－159，161，164，
189－190，248
sqlite3_finalize() function（sqlite3_finalize()函
数），129，161，163,171，224，272
sqlite3_free() function（sqlite3_free()函数），157，
203
sqlite3_free_table() function（sqlite3_free_table()
函数），159
sqlite3_free(errmsg) function（sqlite3_free(errmsg)
函数），157
sqlite3_get_table() function（sqlite3_get_table()
函数），132－133，159－161，164，189
sqlite3_interrupt() function（sqlite3_interrupt()函
数），189
sqlite3_last_insert_rowid() function
（sqlite3_last_insert_rowid()函数），159
sqlite3_malloc() function（sqlite3_malloc()函数），
159，202
sqlite3_mprintf() function（sqlite3_mprintf()函
数），134，171，203
sqlite3_open() function（sqlite3_open()函数），
128，154
sqlite3_open_v2() function（sqlite3_open_v2()函
数），154－155，174,271
sqlite3_open_v2()，sqlite3_prepare()
method（sqlite3_open_v2()，sqlite3_prepare()方
法），263
sqlite3_open16() function（sqlite3_open16()函
数），154
sqlite3_prepare() function（sqlite3_prepare()函
数），163－164，169－170，172－173，
178，228
sqlite3_prepare()_v2 function
（sqlite3_prepare()_v2 函数），131
sqlite3_prepare_v2() function
（sqlite3_prepare_v2()函数），129，131，
161－162，164，174，202，223
sqlite3_progress_handler() function
（sqlite3_progress_handler()函数），
189,237
sqlite3_release_memory() function
（sqlite3_release_memory()函数），193
sqlite3_reset() function（sqlite3_reset()函数），
132，161，163，171,229，235
sqlite3_result_error() function（sqlite3_result_

error()函数），203
sqlite3_result_error_nomem() function（sqlite3_result_error_nomem()函数），208
sqlite3_result_text() function（sqlite3_result_text()函数），202－203，208
sqlite3_result_value() function（sqlite3_result_value()函数），203
sqlite3_result_xxx() function（sqlite3_result_xxx()系列函数），203，208
sqlite3_rollback_hook() function（sqlite3_rollback_hook()函数），135，179
sqlite3_set_authorizer() function（sqlite3_set_authorizer()函数），135,176，180，188，237
sqlite3_soft_heap_limit() function（sqlite3_soft_heap_limit()函数），193
sqlite3_step() function（sqlite3_step()函数），129，161－162，170－171，177－178，192，223，228
sqlite3_stmt handle（sqlite3_stmt 句柄），127，129，227
sqlite3_stmt structure（sqlite3_stmt 结构），161，170，233－234，248
sqlite3_trace() function（sqlite3_trace()函数），178，201，237
sqlite3_unlock_notify() function（sqlite3_unlock_notify()函数），192
sqlite3_update_hook() function（sqlite3_update_hook()函数），135，179
sqlite3_user_data() function（sqlite3_user_data()函数），198
sqlite3_value_blob() function（sqlite3_value_blob()函数），199
sqlite3_value_bytes() function（sqlite3_value_bytes()函数），199
sqlite3_value_text() function（sqlite3_value_text()函数），208
sqlite3_value_type() function（sqlite3_value_type()函数），199
sqlite3_value_xxx() functions（sqlite3_value_xxx()系列函数），198－199
sqlite3_vmprintf() method（sqlite3_vmprintf()方法），240
sqlite3_wal_autocheckpoint() function（sqlite3_wal_autocheckpoint()函数），320－321
sqlite3_wal_checkpoint() function（sqlite3_wal_checkpoint()函数），320－321
sqlite3_wal_hook() function（sqlite3_wal_hook()函数），320
sqlite3BtreeBeginStmt routine（sqlite3BtreeBeginStmt 函数），309
sqlite3BtreeBeginTrans routine（sqlite3BtreeBeginTrans 函数），308
sqlite3BtreeClearTable routine（sqlite3BtreeClearTable 函数），309
sqlite3BtreeClose routine（sqlite3BtreeClose 函数），308
sqlite3BtreeCloseCursor function（sqlite3BtreeCloseCursor 函数），309
sqlite3BtreeCommit routine（sqlite3BtreeCommit 函数），308
sqlite3BtreeCommitStmt routine（sqlite3BtreeCommitStmt 函数），309
sqlite3BtreeCreateTable routine（sqlite3BtreeCreateTable 函数），309
sqlite3BtreeCursor function（sqlite3BtreeCursor 函数），309
sqlite3BtreeData function（sqlite3BtreeData 函数），310
sqlite3BtreeDataSize function（sqlite3BtreeDataSize 函数），310
sqlite3BtreeDelete function（sqlite3BtreeDelete 函数），310

sqlite3BtreeDropTable routine
（sqlite3BtreeDropTable 函数），309
sqlite3BtreeFirst function（sqlite3BtreeFirst 函数），
309
sqlite3BtreeGetAutoVacuum function
（sqlite3BtreeGetAutoVacuum 函数），310
sqlite3BtreeGetPageSize function
（sqlite3BtreeGetPageSize 函数），310
sqlite3BtreeInsert function（sqlite3BtreeInsert 函
数），310
sqlite3BtreeKey function（sqlite3BtreeKey 函数），
310
sqlite3BtreeKeySize function（sqlite3BtreeKeySize
函数），310
sqlite3BtreeLast function（sqlite3BtreeLast 函数），
309
sqlite3BtreeMoveto function（sqlite3BtreeMoveto
函数），309
sqlite3BtreeNext function（sqlite3BtreeNext 函数），
309
sqlite3BtreeOpen routine（sqlite3BtreeOpen 函数），
308
sqlite3BtreePrevious function（sqlite3BtreePrevious
函数），309
sqlite3BtreeRollback routine（sqlite3BtreeRollback
函数），309
sqlite3BtreeRollbackStmt routine
（sqlite3BtreeRollbackStmt 函数），309
sqlite3BtreeSetAutoVacuum function
（sqlite3BtreeSetAutoVacuum 函数），310
sqlite3BtreeSetBusyHandler function
（sqlite3BtreeSetBusyHandler 函数），310
sqlite3BtreeSetCacheSize function
（sqlite3BtreeSetCacheSize 函数），310
sqlite3BtreeSetPageSize function
（sqlite3BtreeSetPageSize function 函
数），310
sqlite3BtreeSetSafetyLevel function
（sqlite3BtreeSetSafetyLevel 函数），310
sqlite3.def file（sqlite3.def 文件），21，26
sqlite3.dll file（sqlite3.dll 文件），21
sqlite3.exe file（sqlite3.exe 文件），19－20，30
sqlite3.lib library（库文件 sqlite3.lib），21－22，
28
SQLite.Callback interface（SQLite.Callback 接口），
237，239
sqliteCreateAggregate() method
（sqliteCreateAggregate()方法），251
sqliteCreateFunction() method
（sqliteCreateFunction()方法），251
SQLiteDatabase class（SQLiteDatabase 类），
286－290
convenience methods for（便捷方法），288－289
executing queries with（执行查询），287－288
implementing（实现），290－292
opening and closing databases with（打开与关闭
数据库），286－287
transactions with（事务），289
useful methods for（有用的方法），290
SQLiteDatabase::compile() method
（SQLiteDatabase::compile()方法），238
SQLite.Database.create_function()
function（SQLite.Database.create_function()
函数），241
SQLiteDatabase.CursorFactory class
（SQLiteDatabase.CursorFactory 类），285
SQLite.Database::exec() method
（SQLite.Database::exec()方法），240
SQLite.Database.function_type()
function（SQLite.Database.function_type()函数），
241
SQLiteDatabase::open() method
（SQLiteDatabase::open()方法），238
sqlite.fossil file（sqlite.fossil 文件），23

SQLite.Function interface(SQLite.Function 接口)，240 - 241
sqlite.jar file（sqlite.jar 文件），237，241
SQLite.JDBCDriver class（SQLite.JDBCDriver 类），241
SQLiteJDBCExample.java file（SQLiteJDBCExample.java 文件），242
SQLiteJNIExample.java file（SQLiteJNIExample.java 文件），238
SQLiteman tool（SQLiteman 工具），45
SQLiteManager tool（SQLiteManager 工具），46
SQLiteOpenHelper() method(SQLiteOpenHelper() 方法)，287
SQLiteOpenHelper class，for Android development（Android 开发中的 SQLiteOpenHelper 类），285 - 286， 290 - 292
SQLiteQueryBuilder class，for Android development（Android 开发中的 SQLiteQueryBuilder 类），293 - 294
sql.sql file（sql.sql 文件），49
src folder（src 文件夹），25
stable source distribution（稳定的源代码发布版），compiling source code on Windows（Windows 上从源代码编译），22
stackable flags parameter(可堆栈化的标志参数)，287
startup（启动），accessing database upon（访问数据库），269 - 272
Statement::bind_params() method（Statement::bind_params()方法），234
Statement.close() method（close()方法），234
statements，data structures of SQLite（SQLite 的语句，数据结构），126 - 127
Statically linked command-line program(静态链接的命令行程序)，31
step() function（step()函数），131，137，146 - 148，151,198，231，241
storage classes for SQLite（SQLite 中的存储类），101 - 104
and type affinity（类型亲缘），314
and type conversions（类型转换），315 - 318
str_agg() function（str_agg()函数），206 - 207，209
str_agg_finalize() function（str_agg_finalize()函数），206 - 208
str_agg_step() function(str_agg_step()函数)，206 - 208
strcat() function（strcat()函数），206
subqueries（子查询），79 - 80
subselects（子选择），79
sum() function（sum()函数），67，137，195
sum_agg() finalize function（sum_agg()结束函数），208
Synaptic utility（Synaptic 程序），31
synchronous pragma(编译指示 同步)，121 - 122，143
syncs() function（syncs()函数），310
syntax for SQL in SQLite（SQLite 中的 SQL 语法），50 - 53
 commands（命令），51 - 52
 comments（注释），53
 identifiers（标示符），53
 keywords（关键字），53
 literals（字符），52
System Variables list（系统变量列表），19

■T

t parameter（参数 t），266
table functions，for B-tree module API（B-tree 模块 API 的表函数），309
table parameter（参数 table），293
table_info pragma（编译指示 table_info），121，242
table_name attribute（属性 table_name），54

table_name variable（变量 table_name），106
table_name.column_name notation
（table_name.column_name 注释），73
Tables（表）
aliases for（别名），77－79
altering（修改），54－55
creating（创建），53－54
joining（连接），72－77
cross joins（交叉连接），75－76
inner joins（内连接），74－75
natural joins（自然连接），77
outer joins（外连接），76
preferred syntax for（优先语法），77
.tables [pattern] command（.tables [pattern]命令），
37
tables variable（变量 tables），58
TakeOffGW network installer（TakeOffGW 网络安
装），28－29
TakeOffGW package（TakeOffGW 包），28－30
tar -xzvf sqlite-amalgamation-3.6.23.1.tar.gz
command（tar -xzvf sqlite-amalgamation-
3.6.23.1.tar.gz 命令），30
Tcl
language extension for（Tcl 语言扩展），
243－247
connecting to（连接），244
installation of（安装），243－244
query processing with（查询处理），244－246
user-defined functions with（用户自定义函
数），247
parameters in C API（C API 中的参数），
173－174
Tcl Extension Architecture (TEA)（Tcl 扩展架构），
18
tclsqlite.c file（tclsqlite.c 文件），26
TEA (Tcl Extension Architecture)（Tcl 扩展架构），
18
temp database（临时数据库），118
temp keyword（关键字 temp），54
temp_store pragma（编译指示 临时存储），
122，131
temporary keyword（关键字 temporary），54
temporary table（临时表），54
test table（test 表），36－38，40－42，114
test_idx index（test_idx 索引），38
test2 table（test2 表），41
test2.db database（test2.db 数据库），42
test3.db database（test3.db 数据库），42
test.csv file（test.csv 文件），41
test.db database（test.db 数据库），35，42
test.db file（test.db 文件），36，39，43
test.sql file（test.sql 文件），42－43，49
text class（text 类），102－103，196
Text Field elements（文本编辑框元素），267
threading（线程）
with SQLite API（SQLite API），136
support in C API（C API 中的支持），190－193
and memory management（内存管理），193
shared cache mode（共享缓存模式），
190－193
THREADSAFE preprocessor flag（线程安全预处
理标示），21
three-value logic（三值逻辑），85
/tmp folder（/tmp 文件夹），33
tokens（符号），52
Transaction method（事务方法），246
transactions（事务），111－118，138－149
and conflict resolution（冲突解决），112－115
and deadlocks（死锁），116－117
and exclusive state（独占状态），145
life cycles for（生命周期），138－139
lock states（锁定状态），139－141
and locks（锁），115－116
read transactions（读操作事务），141

scope for（范围），111 - 112
sizing page cache，（调整页面缓存大小）145 - 146
with SQLiteDatabase class（用 SQLiteDatabase 类），289
types of（类型），117 - 118
using busy handler（繁忙处理），146 - 147
write transactions（写操作事务），141 - 145
and autocommit（自增长），143 - 145
exclusive state（独占状态），142 - 143
and page cache（缓存页面），145
pending state（待定状态），142
reserved state（保留状态），141 - 142
triggers（触发器），108 - 111
error handling with（错误处理），110
and updatable views（可更新的视图），110 - 111
update triggers（更新触发器），109 - 110
tristate logic（三态逻辑），85
type affinity（类型亲缘），313 - 315
and column types（字段类型），313
example of（示例），314 - 315
and storage classes（存储类），314
type conversions，and storage classes（类型转换，存储类），315 - 318
type_id attribute（type_id 属性），48，65，68，71
type_id column（type_id 列），72，76，87 - 88，100
type_id group（type_id 组），68
type_id values（type_id 值），68，71 - 72，83，89
typeof() function（typeof()函数），102，317
typeof(b) column（typeof(b)列），315
types() method（types()方法），239
types subquery（types 子查询），80

■U

UIViewController class（UIViewController 类），266
unary operators（一元操作符），60
union all operation（union all 操作），81
union keyword（关键字 union），81
union operation（联合操作），81
UNION statement（UNION 语句），12
unique constraints（唯一性约束），54，91 - 93，96 - 97，106,112 - 113，115
unique keyword（关键字 unique），106
unlock notification（解锁通知），shared cache mode（共享缓存模式），192 - 193
UNLOCKED state（未锁定状态），140 - 144
UnlockFile() function（UnlockFile()函数），148
updatable views，and triggers（可更新的视图，触发器），110 - 111
update command（更新命令），91 - 92，98，108，115,141 - 142，145，147，149
update filter（更新过滤器），183
update operation（更新操作），135，179 - 180
UPDATE statement（更新语句），10 - 12，110，112
update triggers（更新触发器），109 - 110
update_hook() function，in C API（C API 中的 update_hook()函数），179 - 180
update_list column assignment（指定更新字段的列表），91
upper() function（upper()函数），66
UPSERT statement（UPSERT 语句），10
USB Driver for Windows component（Windows 上 USB 驱动组件），282
user interface，linking data with（连接数据与接口），298 - 299
User Interfaces group（用户接口组），266
user-defined functions，with language extensions（扩展语言中的用户自定义函数）
for Java（Java），240 - 241
for Perl（Perl），224 - 225

for PHP（PHP），251 - 252
for Python（Python），231
for Ruby（Ruby），236
for Tcl（Tcl），247
utilities（实用程序），8

V

vacuum command（命令），119，122
VACUUM operation（VACUUM 操作），189
value argument（参数 value），199
value field（value 域），36，41
value_list variable（变量 value_list），87
values（值）
for Extension C API（扩展 C API），198 - 199
for where clause（where 子句），60
varchar(100) column（varchar(100)列），9
VDBE (virtual database engine)（虚拟数据库引擎），6，161
Venn diagram（纬恩图），74
version parameter（参数 version），285
views，in SQLite（SQLite 中的视图），104 - 106
virtual database engine (VDBE)（虚拟数据库引擎），6，161
VM (virtual machine)（虚拟机），6 - 7，238
VM::step() method（VM::step()方法），239
void method（void 方法），288

W

WAL (Write Ahead Log)（预写日志），318 - 322
activation of（激活），319
advantages and disadvantages of（优缺点），320
and checkpoints（检查点），318
and concurrency（并发性），319
configuration of（配置），320
operational issues with（操作问题），321 - 322
overview（纵览），318
when condition（when 情况），83 - 84
where clause（where 子句），59 - 64
binary operators（二进制操作符），60 - 62
glob operator（glob 操作符），64
like operator（like 操作符），63
logical operators（逻辑操作符），63
operators（操作符），60
values（值），60
where keyword（where 关键字），57
where predicate（where 断言），84
while loop（while 循环），149
.width option（.width 项），49
Win 32 Application Wizard（Win32 应用程序向导），28
Win32 Project template（Win32 工程模板），27
Windows，18 - 30
building dynamically linked SQLite client with Visual C++（Visual C++编译动态链接 SQLite 客户端），27 - 28
building SQLite DLL with Microsoft Visual C++（Visual C++编译 SQLite DLL），25 - 27
building SQLite with MinGW（MinGW 编译 SQLite），28 - 30
compiling SQLite source code on Windows（Windows 上编译 SQLite 源代码），22 - 25
installing command-line program（安装命令行程序），18 - 21
installing SQLite DLL（安装 SQLite DLL），21 - 22
Windows command shell（Windows 命令行 shell），19
\windows\system32 folder（\windows\system32 文件夹），19
--with-jardir=DIR option（--with-jardir=DIR 项），237
wrapped queries，executing（封装查询，执行），132 - 133

wrapper functions，in C API（C API 中的封装函数），153 - 160
for connecting and disconnecting（连接与断开），153 - 155
exec() function（exec()函数），155 - 159
get_table() function（get_table()函数），159 - 160
Write Ahead Log. *See* WAL（预写日志，参考 WAL）
write transactions（写操作事务），141 - 145
and autocommit（自增长），143 - 145
exclusive state（独占状态），142 - 143
and page cache（缓存页面），145
pending state（待定状态），142
reserved state（保留状态），141 - 142
Write-Ahead Log mode（预写日志模式），192

X

xCallback function（函数 xCallback），179
xcode_3.2.4_and_ios_sdk_4.1.dmg file（xcode_3.2.4_and_ios_sdk_4.1.dmg 文件），254
xCompare argument（参数 xCompare），213
xFinal argument（参数 xFinal），197，206
xFunc argument（参数 xFunc），197，206
xNotify function（函数 xNotify），192
xStep argument（参数 xStep），197，206

Y

<yourdb>-shm file（你的数据库名>-shm 文件），320
<yourdb>-wal file（<你的数据库名>-wal 文件），320

Z

zero configuration，features of SQLite（SQLite 的零配置特性），8
zFunctionName argument（参数 zFunctionName），197
zVfs parameter（参数 zVfs），155

[General Information]
书名=SQLite权威指南　　第2版
作者=（美）艾伦,（美）欧文斯著
页数=380
出版社=北京市：电子工业出版社
出版日期=2012.01
SS号=12914460
DX号=000008214855
URL=http://book.szdnet.org.cn/bookDetail.jsp?dxNumber=000008214855&d=9D585951758AD2326DA986D58A17B7E8